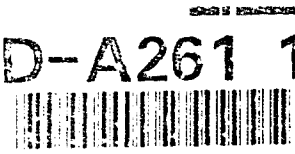
AD-A261 198



0

TECHNICAL REPORT ITL-92-9

INTELLIGENT ACCESS TO MULTIPLE RELATIONAL DATABASE SYSTEMS

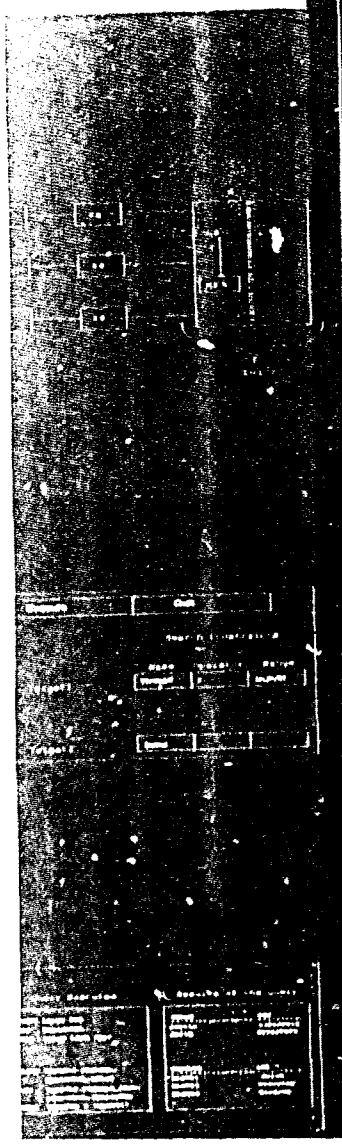
by

Peggy Wright

Information Technology Laboratory

DEPARTMENT OF THE ARMY
Waterways Experiment Station, Corps of Engineers
3909 Halls Ferry Road, Vicksburg, Mississippi 39180-6199

US Army Corps of Engineers

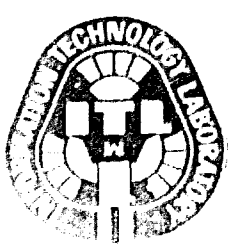


November 1992
Final Report

Approved For Public Release; Distribution Is Unlimited

93-03981

DTIC
ELECTE
FEB 26 1993
S E D



Prepared for Discretionary Research Program
US Army Engineer Waterways Experiment Station
3909 Halls Ferry Road, Vicksburg, Mississippi 39180-6199

Destroy this report when no longer needed. Do not return it to the originator.

The findings in this report are not to be construed as an official Department of the Army position unless so designated by other authorized documents.

The contents of this report are not to be used for advertising, publication, or promotional purposes. Citation of trade names does not constitute an official endorsement or approval of the use of such commercial products.

REPORT DOCUMENTATION PAGE

Form Approved
OMB No 0704-0188

Public reporting burden for this collection of information is estimated to average 1 hour per response, including the time for reviewing instructions, searching existing data sources, gathering and maintaining the data needed, and completing and reviewing the collection of information. Send comments regarding this burden estimate or any other aspect of this collection of information, including suggestions for reducing this burden to Washington Headquarters Services, Directorate for Information Operations and Reports, 1215 Jefferson Davis Highway, Suite 1204, Arlington, VA 22202-4302, and to the Office of Management and Budget, Paperwork Reduction Project (0704-0188), Washington, DC 20503

1. AGENCY USE ONLY (Leave blank)	2. REPORT DATE	3. REPORT TYPE AND DATES COVERED
	November 1992	Final report

4. TITLE AND SUBTITLE

Intelligent Access to Multiple Relational Database Systems

5. FUNDING NUMBERS

6. AUTHOR(S)

Peggy Wright

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)

US Army Engineer Waterways Experiment Station
Information Technology Laboratory
3909 Halls Ferry Road, Vicksburg, MS 39180-6199

8. PERFORMING ORGANIZATION REPORT NUMBER

Technical Report ITL-92-9

9. SPONSORING / MONITORING AGENCY NAME(S) AND ADDRESS(ES)

Discretionary Research Program
US Army Engineer Waterways Experiment Station
Information Technology Laboratory
3909 Halls Ferry Road, Vicksburg, MS 39180-6199

10. SPONSORING / MONITORING AGENCY REPORT NUMBER

11. SUPPLEMENTARY NOTES

Available from National Technical Information Service, 5285 Port Royal Road, Springfield, VA 22161.

12a. DISTRIBUTION / AVAILABILITY STATEMENT

Approved for public release; distribution is unlimited.

12b. DISTRIBUTION CODE

13. ABSTRACT *(Maximum 200 words)*

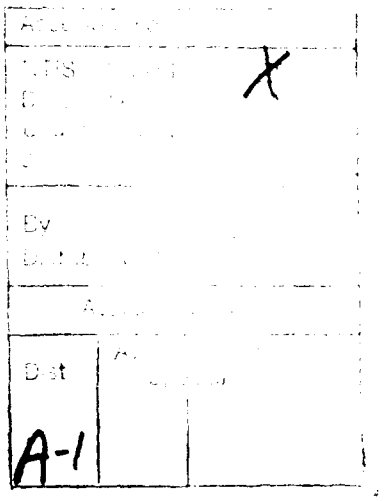
The combination of artificial intelligence (AI) and database technology allows the capability of extending the usefulness of existing database applications. More specifically, the use of AI provides the means to develop an intelligent front end to allow nonintrusive assess to multiple relational databases. A knowledge source is built for each participating database application. A meta-knowledge source, constructed from all knowledge sources in the system, is consulted to determine the appropriate application to which a database request is channeled. This report introduces a proposed architecture and presents a prototype developed to accomplish this task.

14. SUBJECT TERMS	15. NUMBER OF PAGES
See reverse.	175
	16. PRICE CODE

17. SECURITY CLASSIFICATION OF REPORT	18. SECURITY CLASSIFICATION OF THIS PAGE	19. SECURITY CLASSIFICATION OF ABSTRACT	20. LIMITATION OF ABSTRACT
UNCLASSIFIED	UNCLASSIFIED		

14. (Concluded).

Composite databases	Federated databases	Interconnectivity
Co-relationship	Heterogeneous database systems	Multidatabase systems
Distributed databases	Homogenous database systems	Relational database management systems



PREFACE

This technical report documents a proposed design architecture and methodology used to integrate multiple relational database systems, in support of the Intelligent Access to Multiple Relational Databases project under the Discretionary Research Program. The proposed architecture incorporates the use of artificial intelligence in an intelligent front end to provide non-intrusive access to multiple relational database systems. The intelligent front end is comprised of a user interface, including a meta-knowledge source, and individual knowledge sources for each database application.

This research and development was done and this report was written at the US Army Engineer Waterways Experiment Station, Information Technology Laboratory (ITL), Computer Science Division, Systems Modernization Unit, by Ms. Peggy Wright. The work was sponsored by funds provided by the Discretionary Research Program. The skillful coding efforts of Mr. Shannon Thornton are gratefully acknowledged. His patience and painstaking effort in following specification to minute detail helped make this research effort possible.

Ms. Mary K. Vincent, Chief, Office of Technical Programs and Plans, WES, is manager of the Discretionary Research Program. The work was accomplished at WES under the supervision of Ms. Barbara Comes, Chief, Systems Modernization Unit, Dr. Windell Ingram, Chief, Computer Science Division, and Dr. N. Radhakrishnan, Director, ITL.

At the time of publication of this report, Director of WES was Dr. Robert W. Whalin. Commander was COL Leonard G. Hassell, EN.

CONTENTS

		Page
PREFACE		i
1	Introduction	1
2	Background	2
3	Design Architecture	4
	3.1 Intelligent User Interface Components	8
	3.2 Knowledge Source Components	9
4	Implementation Methodology	11
	4.1 Assumptions and Constraints	12
	4.2 Development Environment	13
	4.3 Prototype Databases	13
	4.4 Knowledge Source Components	14
	4.5 Knowledge Source Implementation	20
	4.6 Intelligent User Interface Components	21
	4.7 Intelligent User Interface Implementation	24
	4.8 Implementation Advantages	25
	4.9 Future Work	27
5	Conclusions	29
6	References	31
7	Appendices	
	A. Application Database Design	33
	B. SQL Code	37
	C. PRO*C Code	71
	D. C Code	101

FIGURES

1.a Intelligent Access Architecture Design 5

1.b Refined Intelligent Access Architecture Design 7

2 Intelligent User Interface Design 8

3 Knowledge Source Design 9

4.a Knowledge Source Table Frame Structure 17

4.b Knowledge Source Column Frame Structure 17

4.c Knowledge Source Index Frame Structure 18

5.a User Interface Input Window 22

5.b User Interface SQL Display Window 22

5.c User Interface Query Results Window 22

6 Meta-knowledge Source Column Frame Structure 24

A.1 Personnel Database Relational Schema 35

A.2 Personnel Database Entity Relational Diagram 35

A.3 Vehicle Database Relational Schema 36

A.4 Vehicle Database Entity Relational Diagram 36

1 Introduction

There are many applications employing databases currently in use. Consequently, a single access point to multiple databases is of significant interest because it allows information resources to be shared without undergoing additional development effort. Many terms such as *multidatabase systems*, *federated databases*, *composite databases*, and *distributed databases* have been introduced in recent years to express this concept.

> "Multidatabase systems give users a common interface to multiple databases, while minimizing the impact on existing database operations" (Bright, et al., 1992).
>
> "A distributed database is a collection of multiple, logically interrelated databases distributed over a computer network" (Özsu and Valduriez, 1992).

The major difference in multidatabase systems and distributed databases is that multidatabase systems consist of heterogeneous systems with different user access methods. A distributed database has only one database management system (DBMS) and therefore one set of access methods, but consists of multiple distributed database nodes on a network. While there is much discussion in this area, the commercial technology is still lagging.

Many conceptual models have been developed in an attempt to provide access to multiple databases. These include Open Intelligent Information Systems Architecture (Kaula, 1990), and a model discussed in "Using Knowledge-Based Technology to Integrate CIM Databases" (Dilts and Wu, 1991) that presents a conceptual

framework to allow connection of pre-existing heterogeneous databases. This architecture proposed a knowledge-based system to perform "communication and processing of shared information."

The idea of using artificial intelligence (AI) to extend the usefulness of computer systems is not a new one. Since the field's beginning in 1956 (Charniak and McDermott, 1986), AI has grown and become a highly accepted technology.

AI is a common tool that may be used to extend the capabilities of a computer program or system. Historically, these systems run the gamut from applications like an expert system to visually identify microfossils (Swaby, 1992), to speech recognition (Nii, 1986), to a sophisticated targeting system (Dunn, 1990). Other uses for AI include the field of robotics (Meiran, 1988), voice recognition (DeYoung, 1984) and terrain analysis (Maune, 1991). The previously mentioned conceptual models developed to access multiple databases all propose the use of AI to achieve this extended access.

2 Background

The primary goal of Intelligent Access to Multiple Relational Database Systems (IAMS) is to investigate, design, and develop a prototype model to provide nonintrusive intelligent access to multiple databases. Nonintrusive implies that a) the database design does not have to be altered in any manner; b) access is query only, i.e., no updates; and c) impact on participating databases is minimized. Intelligent access is the ability to determine which participating database or databases

has the target data and retrieve it. Target data is defined as the data needed to satisfy a user request. Relational database theory is the basis for the database techniques and analysis methodology used in this research.

Intelligent access is especially needed when co-relationships exist among databases. When data from more than one source are required to respond to a request, then a simple co-relationship exists. When information from one query is used to form additional queries in responding to a request, then a complex co-relationship exists.

With most multiple database or distributed database applications, the user or the user interface must know which database has the target information. The intelligent aspect or intent of IAMS is to have the user interface dynamically determine and query any database containing the target data.

This capability extends the usefulness of existing databases by allowing relationships to be established across databases dynamically at run time while requiring little knowledge on the part of the user. This enhancement means that, for example, a user wanting to know information about a particular site could find out about personnel and equipment at the site with one query. Without this enhancement, the example might require two separate queries or even phone calls to request different people to query two separate databases.

Databases are continuously being redesigned and expanded to add new related data. Through the use of AI, database

application systems can be extended and used in ways not previously considered without structural database redesign. These additional resources can be achieved without expensive modification to the underlying database applications, thus extending the usefulness of existing database systems by providing multiple database access. This concept known as "interconnectivity" is a means of sharing information between arbitrary database systems (Brodie, 1988). An automated interconnected network of databases introduces the potential to exploit relationships among separate databases to provide increased usefulness.

For more information on the subject of extending the usefulness of existing database systems with AI technology, the user is referred to a book on the subject, *Intelligent Databases* (Parsaye, et al., 1989), and a paper "Potential Methodologies of Intelligent Access to Multiple Databases" (Wright, 1992).

3 Design Architecture

The means of achieving the goal of nonintrusive access to multiple relational databases is through the use of intelligent access. Three plausible designs are explored in detail in "Potential Methodologies of Intelligent Access to Multiple Databases", which drafts the IAMS basic architecture.

Analysis of the three classes of potential access methodologies made apparent that an intelligent front end (IFE) to each existing application database is necessary. Each method explored uses an IFE, whether it is known as a knowledge source

(KS), an expert system, or an autonomous system. The other common link is the communications control mechanism. In most cases this function is incorporated in an intelligent user interface.

A composite architecture derived from the methodologies and applications explored is the basic design which is proposed as a result of this research and is illustrated in Figure 1.a. The intelligence designed into IAMS resides in several modules.

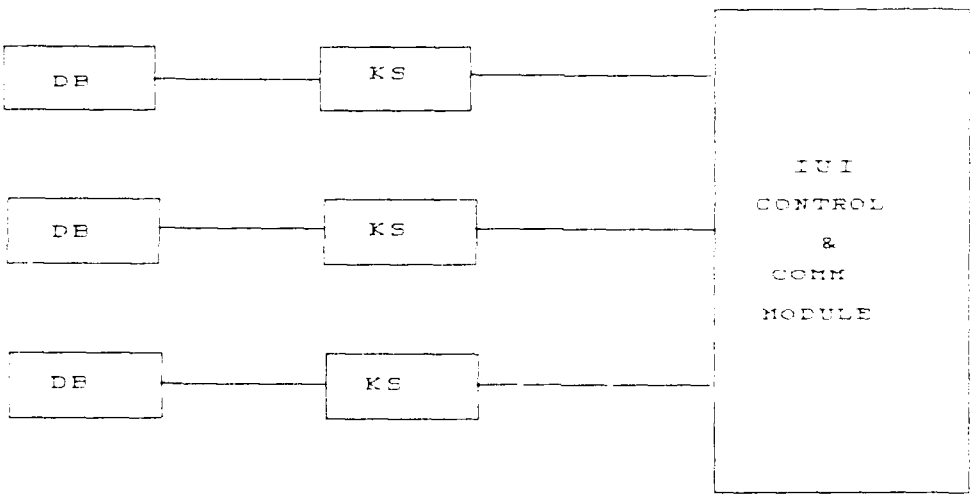


Figure 1.a Intelligent Access Architecture Design

DB$_i$ Existing database system
KS$_i$ Knowledge Source: Knows its database entities/relationships
IUI Intelligent User Interface: Provides user friendly interface, contains mechanisms to control queries, communicates with all KS

The intelligent user interface contains the communications and control mechanisms necessary to communicate user requests to the appropriate KS for translation into a database query. A

separate KS[1] is built for each database that is to be linked into the system. This architecture provides the flexibility to add additional databases to an already functioning network of database systems.

During the design and development of the IAMS prototype, this model was further expanded and refined to the architecture shown in Figure 1.b. The logical functions contained in the IUI are better defined and the modules encapsulated such that the module interfaces became loosely coupled.

The user interacts with the user interface (UI), typing in requests for data and observing the response from the IAMS system. The C&C module receives the request from the UI and relays part of this input to the meta-knowledge source (MKS) which determines the database application or applications that might contain the requested information.

This knowledge is conveyed back to the C&C module which activates a KS process. The KS determines which table or tables potentially contain the necessary data and any related fields that are referenced to find it. A query, generated from this information, is sent to the database. The query results are returned to the C&C which checks the results and passes them to the UI for display to the user screen.

---

[1] Knowledge sources are defined by Penny Nii (Nii, 1986) as "logically independent sources of knowledge."

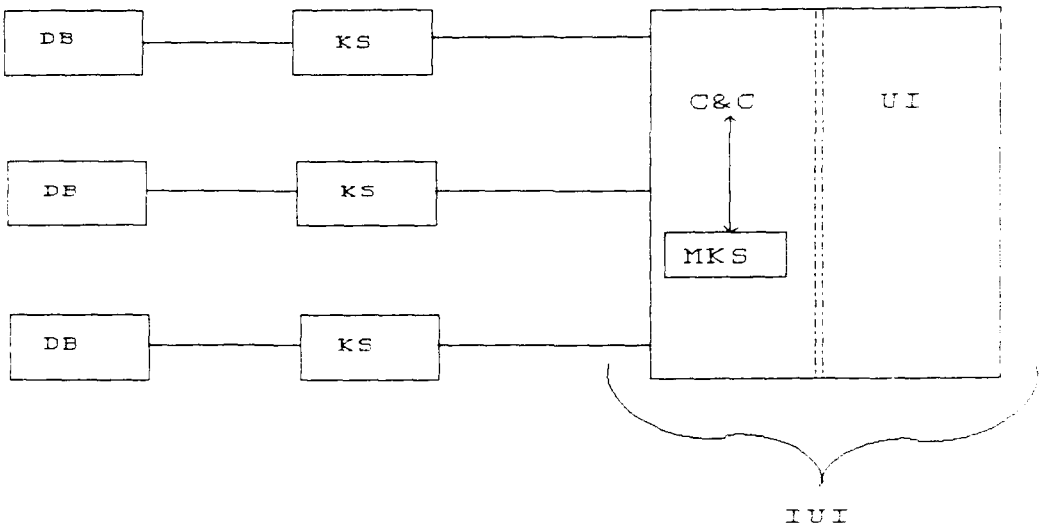


Figure 1.b Refined Intelligent Access Architecture Design

DB$_i$ Existing database system
KS$_i$ Knowledge Source: Knows its database entities/relationships
C&C Communications and Control module: provides communication between UI and KS modules
MKS Meta-knowledge Source: contains knowledge about KSs and relationships between them
UI User Interface: accepts user input, displays results
IUI Intelligent User Interface: Provides user friendly interface, contains mechanisms to control queries, communicates with all KS

As stated previously, the intelligence in IAMS is contained in several modules. The UI uses intelligence to preprocess the user input to determine which database or databases to query. After database determination has occurred, the KS uses code, including prioritization algorithms, to determine which table or tables to query. Additional intelligence is inherent in the dynamic SQL code generator. The next two sections examine in detail the design architecture of the two intelligent components

of IAMS, the IUI and the KS.

3.1 Intelligent User Interface Components

The C&C module, the MKS, and the UI are all parts of the IUI which is illustrated in Figure 2. The two major functions of the UI module are: a) to accept and relay information from the user to the rest of the system, and b) to accept and return information from the system to the user. It has the added requirement to interact with the user in a consistent and helpful manner.

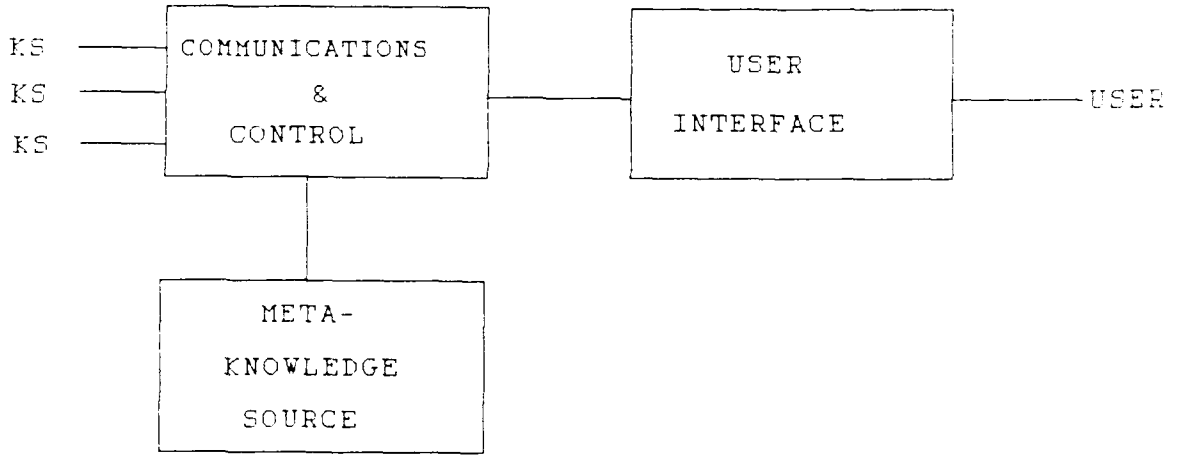


Figure 2 Intelligent User Interface Design

The MKS is an extension or combination of the knowledge contained in all individual KSs. Meta-knowledge is knowledge about knowledge, or more specifically, in IAMS knowledge about multiple knowledge sources. The C&C module has the capability and mission to relay intermediate and final information to the UI

based on observations from the MKS or KS modules. This capability increases the overall functionality and responsiveness of the IAMS system. Intermediate communications take the form of warning or informative messages, or a request for additional qualification from the user based on pre-query processing.

3.2 Knowledge Source Components

Each database in IAMS has a related KS. Each KS contains knowledge or information about the underlying database application and its structure. A KS also contains code which allows it to find appropriate information and determine which table or tables contain a particular data item. Additional code is used to generate specific database queries and send them to the database application. The components of a KS are presented in Figure 3.

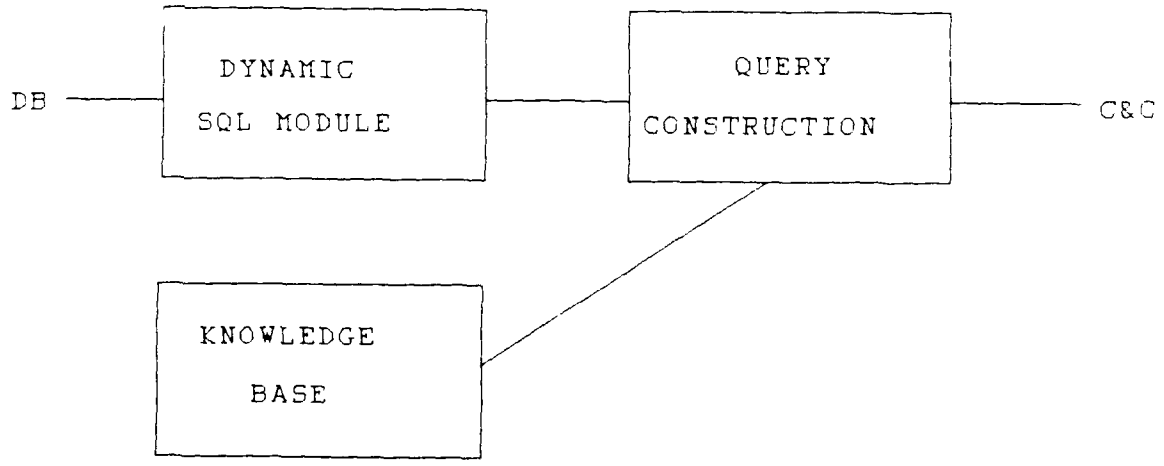


Figure 3 Knowledge Source Module

An active *knowledge base* primarily consists of frames which contain knowledge about the database structure. Each frame contains an entire cross reference of the database for one data element. For example, a frame containing data about the data element SSN should actually be a view of the database according to SSN. It contains information about the data type and size of SSN, and it also contains information about whether SSN can be null[2], whether it is a key field, and names of all the tables SSN occurs in. There are three types of frames stored in each knowledge base: table, column, and index.

The first time a KS is activated the knowledge base frames are loaded from Oracle tables which are built by the initial KS construction process. The tables contain primarily the same information the frames contain, but in a different format. The frames are designed for efficiency in access and a much more complex organizational structure. It is possible to cross reference among column, table, and index frames.

The function performed by the *query constructor* (QC) is self-explanatory. Input passed from the C&C module activates the QC which contains the functionality to build an SQL query. Because the prototype databases are assumed to be homogeneous (i.e., each developed using Oracle), the QC is the same for all databases. The QC depends on information contained in the knowledge base to determine which table or tables contain the

---

[2] A null field is one that is allowed to remain blank or empty. This means the data value a) does not exist; b) is not meaningful for the particular record; or c) is unknown.

target data and if a relation must be referenced to perform the query.

The *dynamic SQL module* (DSQL) is also the same for all Oracle databases. It receives the constructed query and connects to the appropriate database. It translates the query as necessary, queries the database, and returns data (if data found), or a message (if no data found or error occurred) back to the QC.

The DSQL must be able to handle queries only, but a multitude of different queries may conceivably be processed. Any legal SQL query can be processed by the DSQL, and it is limited only by the intelligence of the QC construction algorithm.

4 Implementation Methodology

Intelligent access is gained through a three phase process. In Phase I, the MKS is searched to determine which database application has the target data. The user input is then passed to the application KS and, in Phase II, the KS is searched to determine target table(s). In Phase III, the SQL query is actually constructed and the database dynamically queried. In this scenario, intelligent access means that the end user does not have to know where the data resides.

IAMS implementation is accomplished through the use of several high level languages. Much of the code used to accomplish this project is written in the C programming language. Two other languages are used to access the database, SQL and Pro*C (Oracle Corporation), and Unix script files are used as

necessary for utilitarian purposes.

SQL is a standard high-level database query language which is used for a number of relational DBMS including Oracle (Elmasri and Navathe, 1989). Pro*C is a C language precompiler published by Oracle Corporation, which is used specifically to access Oracle tables using a C interface with embedded SQL statements. Using a Pro*C/C interface adds functionality that can not be obtained by using generic SQL operations alone.

The chosen IAMS window interface is Motif. The original interface was written in X, and then changed to Motif for practical reasons. The Motif window manager adds many predefined user friendly functions that are not readily available in X window manager.

4.1 Assumptions and Constraints

For the purpose of developing a prototype, the following assumptions and constraints are made.

DATABASES ARE IMPLEMENTED IN ORACLE (HOMOGENEOUS ORACLE DATABASES).
STANDARD NAMING CONVENTIONS ARE USED FOR ALL DATABASE ELEMENTS.
DATABASE ENTITIES AND RELATIONSHIPS ARE KNOWN.
PROTOTYPE MODEL WILL ALLOW QUERY ONLY, I.E., NO UPDATES ALLOWED.

By working with homogeneous databases, the underlying methodology can be developed using one set of procedures to interface with the database applications. Heterogeneous databases require separate interface routines for each DBMS type accessed.

Standard naming conventions provide a quick and accurate means of inferring inter-database relationships and intra-

database relationships. The major premise[3] of the naming standards is that each database element having the same name must refer to the same thing, and that two unlike elements may not have the same name.

Database entities and relationships are known because the application expert and the developer are the same, saving development time during prototyping. To integrate an existing application database into IAMS, the application expert would be consulted to confirm knowledge about the application that is stored in the IAMS knowledge base.

The principle behind nonintrusive database integration is that the application database is not adversely affected, and that the access is query only. IAMS implementation fully supports this.

4.2 Development Environment

The IAMS prototype is developed on a Unix-based workstation running SUN OS 4.1A. Oracle version 6.0.30.3.1 DBMS is used as the relational database management system (RDBMS) for the database applications necessary to implement the prototype resulting from this research.

4.3 Prototype Databases

In the prototype system two different but related Oracle databases were created. One contains personnel data and the other contains vehicle data. The relational schema and entity

---

[3] Rule 1, Army Regulation 25-9 specifies that the name of a data element is always the same wherever it occurs.

relationship (ER) diagrams for both the personnel and vehicle applications are given in Appendix A. Data used in both databases are fictitious and for demonstration purposes only.

The Corps of Engineers enforces standard naming conventions (U S Army, 1989) for implementation of new database or information systems, so the IAMS database applications conform to the major premise or Rule 1 of the naming standards as previously explained. Using the standard naming conventions makes analyzing databases easier and provides additional data integrity checking. Other benefits of standard naming conventions will be discussed further in the conclusions section.

4.4 Knowledge Source Components

The KS designed and used for IAMS is a hybrid structure, consisting of relational database tables, knowledge frames, and attached procedures which are written in the C programming language. The process used to construct a KS is refined and automated, thereby providing automated construction of each KS through the use of a generic set of code. This approach essentially means that a KS can be generated for any Oracle database[4] targeted to be accessed by IAMS.

After the target databases are determined, building a knowledge source is the first step to incorporating a database

---

[4] A KS can be automatically generated for any Oracle database that conforms to standard naming conventions. This convention states that any element having the same meaning must have the same name, and the inverse must also be true. If standards are not followed, a synonym list or database mapping must be created preparatory to KS generation.

into IAMS. The KS contains such information as the names of the application tables, columns, and indexes. The KS also captures the relationships between tables and columns.

Several items are required to construct a KS for a target Oracle database application. The application name, database location, and database tablespace name must be known. Resource and connect privilege must be assigned to the user building the KS, and that user must also have access to the application system tables.

Relying on standard naming conventions and Oracle system tables, the basis for the KS is built through a series of SQL scripts. The generic SQL scripts developed to analyze the application tables determine pertinent information about the database application, and create a series of reports that may be used to verify the KS data. These script files and example reports are included in Appendix B.

All database indexes are analyzed to determine if they are primary, foreign, and/or composite keys. Keys or indexes are used for several reasons. A primary reason for indexes is to facilitate the location of a record by creating an index on a field which has a unique value for each record. This type of key is called a primary key.

A foreign key is used to access related data and to check data integrity. A composite key is made up of more than one field (Elmasri and Navathe, 1989), e.g., last and first name could be used a composite key. Information about keys is

analyzed and kept for IAMS to facilitate prioritization of user requests. Locating data through a key is more likely to be successful than with a nonkey column, or a null one.

All database fields (columns) are examined to determine relationships among tables. Statistics are gathered on the number of occurrences of a column name[5], data type, data length, and whether or not a field is allowed to be null. The results are retained in IAMS tables automatically created and populated by SQL scripts. The IAMS tables are the first component of a KS and they are stored in the application tablespace.

Pro*C and C code (as shown in Appendices C and D) is used to extract IAMS information from the created tables to construct frames. There are three frame structures used for a KS: table, column, and index. The structure for each frame type is illustrated in Figures 4.a, 4.b, and 4.c. Basic information that forms each frame is loaded into the frame structure from the tables. These frames or knowledge base form the second component of the KS structure.

A structural view of the KS frames is produced by the KS load procedure. This view may be used by an application expert to verify the KS frame data. Each frame is constructed as a tree and the individual nodes contain both single data elements and doubly linked lists.

---

[5] This provides information on shared columns, which implies relationships.

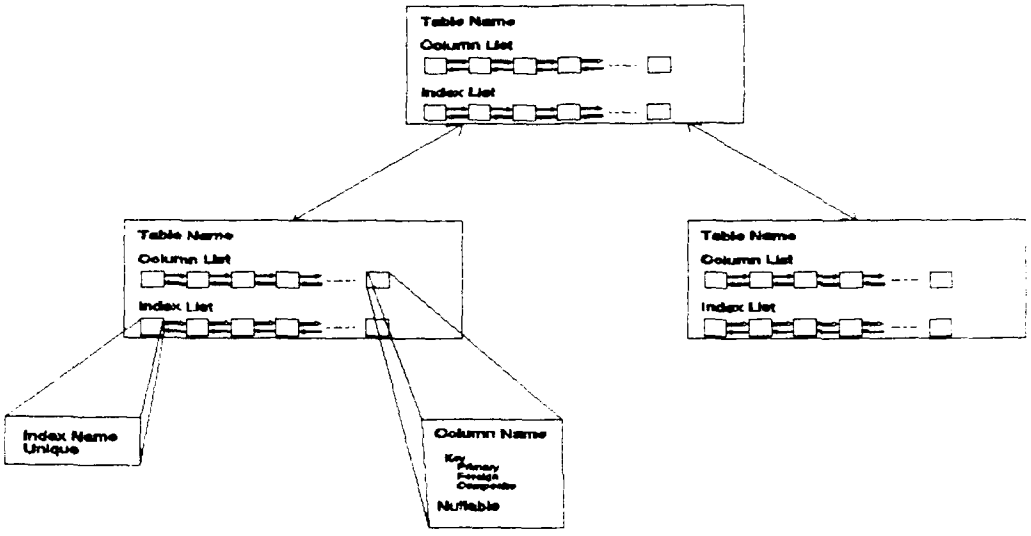


Figure 4.a KS Table Frame Structure

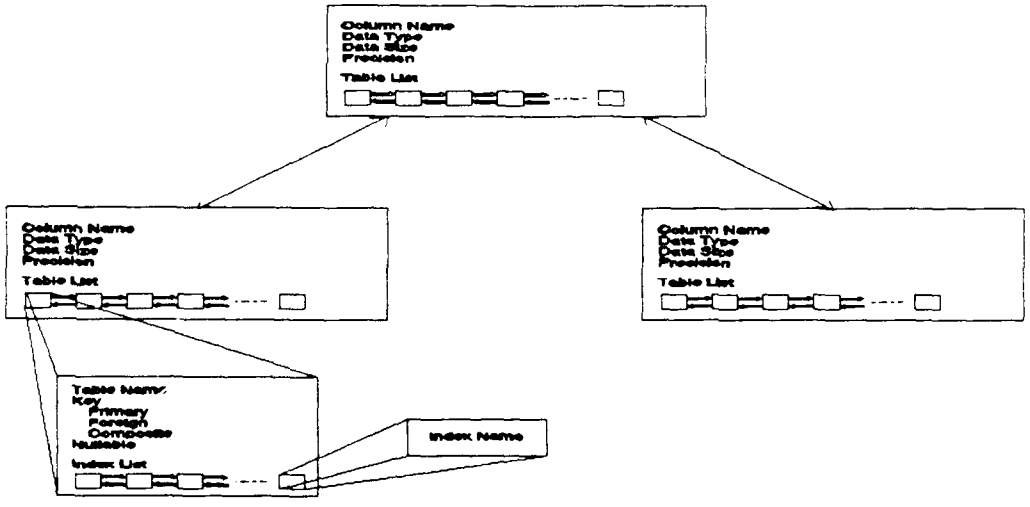


Figure 4.b KS Column Frame Structure

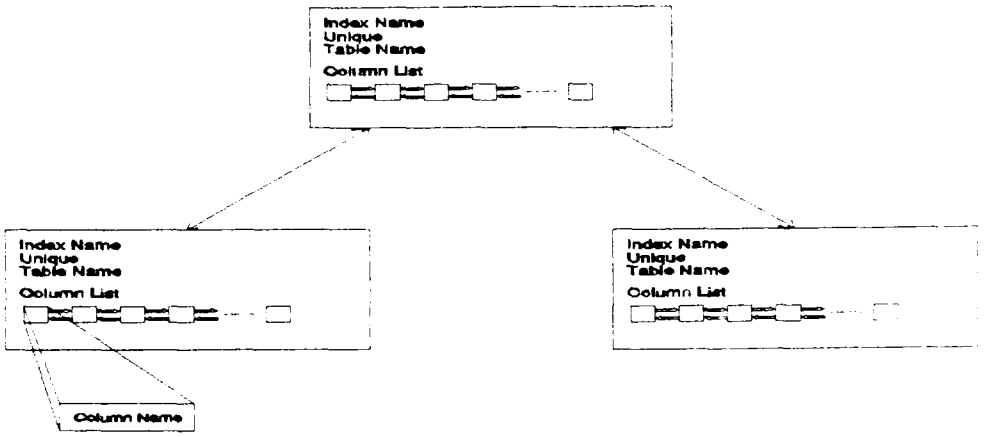


Figure 4.c KS Index Frame Structure

The third component of the KS is also built in C, and attached to the KS frames. These procedural attachments are used to search a related frame type to find information. For example, if the target column is SSN, the column frames are searched until the SSN frame is found. Then data stored in that frame are reviewed to determine in which tables the element SSN occurs.

Data stored in the target column frame are analyzed according to a priority scheme. The KS looks at every potential table containing the target column. If there is more than one table, which contains the target column, the target column characteristics are determined, such as:

> Is the target column a key field, is it a primary key, foreign key, or composite key?

The field search priority is assigned, from highest to lowest, as follows:

primary key
foreign key
single field key
composite key
not null
null allowed

The query being constructed, based on the priority assigned by the search algorithm, is designed to maximize the probability of finding the target element.

The table frames are used to cross-reference field availability, in the event a join column is needed to locate target data. A join column is a field that the target fields both have in common, i.e., a join field appears in the same table with each target, and can be used to build access to the target fields, even though the target fields are not located in a common table.

Two other C modules are necessary to complete the KS functionality. Although not a part of the knowledge base, this code performs two essential functions. The first is the query constructor (QC) which is used to generate an SQL query. The second module is a dynamic SQL procedure (DSQL) which processes the query constructed and actually queries the databases. Both of these modules are DBMS specific.

The QC consists of C code used to generate an SQL query. It accepts input from the C&C and sends names of the target column(s) to the KS search code to locate their KS frames. When

the frames are located, they are searched to determine the appropriate table(s). If a relation must be invoked to access the target data, the potential relations also known as join columns, are displayed to the user so an appropriate choice can be made.

When a table or tables with their related column names are known, they are sent to the QC with any specified search constraints. The QC builds the SQL query and then sends it to the DSQL module, which processes the query and returns results. Results are passed by the QC back to the C&C portion of the IUI.

The DSQL module is written in PRO*C (PRO*C User's Guide, 1990), and accepts an SQL statement from the query constructor. A buffer or descriptor area is prepared to accept the query results from the database, the database is queried, and the results are passed back to the QC.

4.5 Knowledge Source Implementation

When a KS is invoked, it checks for the existence of KS frame data. If the data exist in a related data file, the KS frames are automatically initialized from the file. If the file does not exist, the KS process queries the IAMS tables in the application tablespace and builds the frames. This procedure is followed to facilitate reconstruction of the KS frames if changes are made to the database application structure. The SQL scripts still must be executed to reflect structural database changes, thereby automatically reconstructing the IAMS tables.

This process complies with basic software engineering

principles, because it reduces the number of steps required for regeneration (maintainability), and reduces the potential for error (reliability). This procedure not only facilitates reconstruction if there are structural changes, but also automatically recreates the frame data file if it is erased or missing.

4.6 Intelligent User Interface Components

The IUI is composed of three primary components. The user interface, or UI, is the portion of the system that the user interacts with. The IAMS communication and control functions are contained in the C&C module, which references the primary or meta-knowledge source known as the MKS.

The UI consists of a series of user-friendly windows, that may be manipulated with mouse or keyboard. The input window, shown in Figure 5.a, allows the user to enter query input in a prescribed manner and operates very similarly to a form. Another window, the query display window (Figure 5.b), displays database queries as they are built, and a third, the results window (Figure 5.c), displays the query results. Two example queries are shown in Figure 5.b and their results in Figure 5.c, the input for the last query in shown in Figure 5.a. The window code in contained in Appendix D.

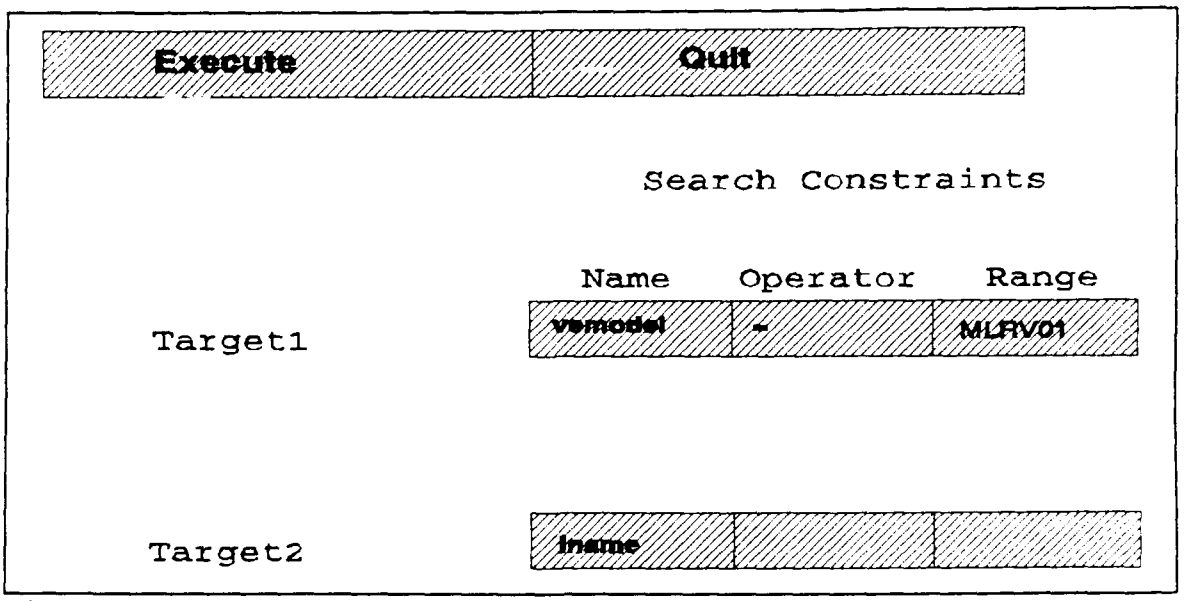


Figure 5.a User Interface Input Window

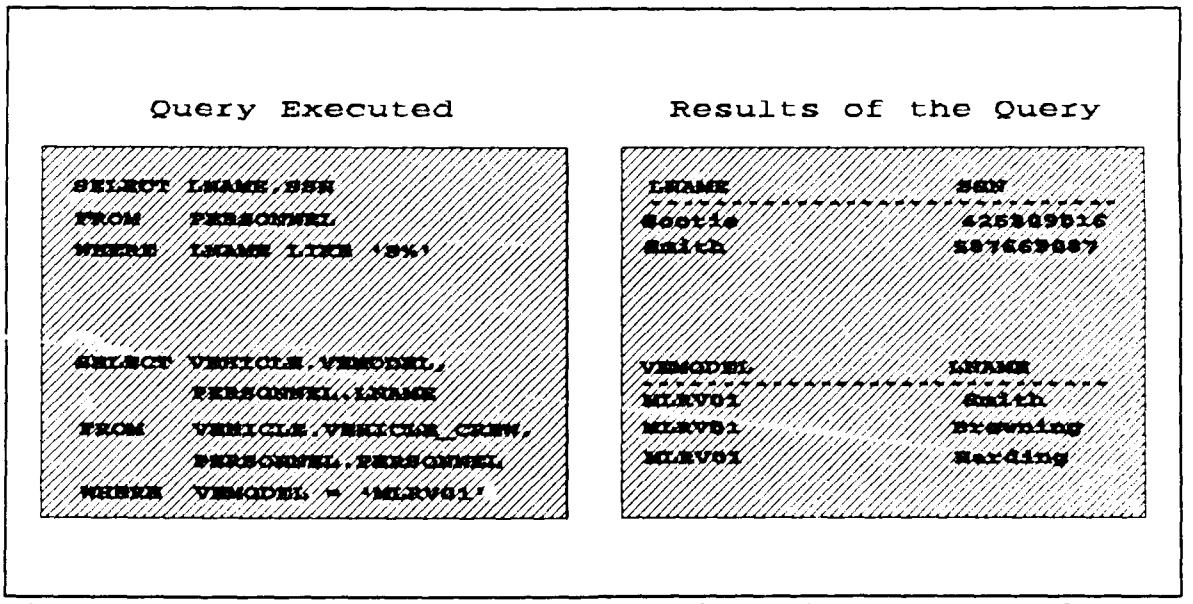


Figure 5.b User Interface SQL Display Window

Figure 5.c User Interface Query Results Window

The prototype version of IAMS allows the user to enter one or two target columns. There is optional search qualification (range condition and operator) allowed for either or both of these. If the target data is located in the same table the results are displayed without further user interaction.

If the target data columns are in different tables or databases, a prioritized search is conducted to determine potential relationships. If more than one potential relationship is found, the user is allowed to choose one from a list and also to enter optional qualification.

The C&C module directs traffic, coordinates with the MKS to determine to which KS to send a request, and receives input from and sends output to the UI. This module also keeps track of which KS has been sent a request, and handles the response as it occurs. Data files are used to store formulated queries and query results, so they can be shared by the different modules. A log file is maintained for the latest user session which provides an audit trail, error tracking, and system evaluation.

The automation of KS construction allows a minimum of effort to add a new database or facilitate changes to current member databases. It also forms the basis for automatic generation of the MKS. The MKS generation is accomplished in a like manner. Similar to the KS, it is also a hybrid structure consisting of the same basic structural parts: tables, frames, and code. The MKS structure is conceptually at a higher level because it contains information about all IAMS database applications and

relationships among them without lower level detail. The MKS is organized in a column frame similar to the KS column frame as presented in Figure 6.

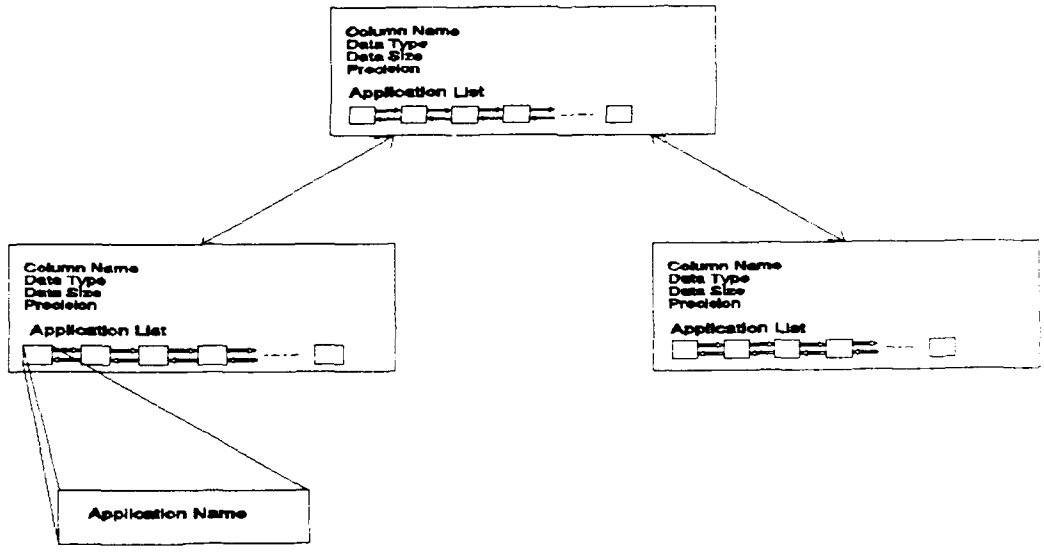


Figure 6 MKS Column Frame Structure

All relationships in a database occur at the column level and that is essentially all that is necessary to determine a relationship when standard naming conventions are employed. Detail such as data type and size is retained in the MKS to allow high level verification of user range variables. The only other MKS information stored is the application name which is necessary to determine which application might have the requested data.

4.7 Intelligent User Interface Implementation

In the current implementation methodology the KS can handle only one query per request. The MKS can process multiple queries, but only one against each KS. The MKS is needed as a

means of determining which KS to channel user requests to. Future adaptations should allow for more flexibility in choosing join columns, and more complex queries, with additional queries based on the results of the previous one, to fully exploit co-relationships.

Each knowledge source has knowledge of a particular database. Knowledge of all databases is necessary at a higher level to be able to determine which database has the target data. Therefore, it is necessary to organize a higher level knowledge source that contains knowledge about the knowledge sources.

The MKS contains meta-knowledge about the KSs and relationships between them. The MKS is built from all KS modules, and is a separate component of the C&C module. The C&C sends a request to the MKS which consists of target column name and the MKS determines which KS(s) to invoke.

The C&C module, while still a part of the IUI, became more clearly delineated during prototype implementation. The primary functions of the UI module are input and display realized through the use of multiple windows. The C&C module performs communication between the UI and the KS modules.

4.8 Implementation Advantages

One of the major implementation advantages of the IAMS prototype is its conformance to software engineering methodology. There are many software engineering principles that apply to the IAMS development process. Most of the code in IAMS is generic and can be used for any candidate database meeting the IAMS

qualifications.

The following list of code is generic, requiring only the database application name for differentiation:

a) SQL database analysis scripts
b) KS construction code
c) Query constructor code
d) Dynamic SQL query procedure
e) Entire KS executable process

Automated construction of individual knowledge bases and MKS is a significant implementation advantage over interactive construction of each KS and MKS.

The IAMS interface was designed to promote future growth and major flexibility in allowing distributed communications to multiple database sites. Although the prototype is developed and demonstrated on one machine, the prototype application databases reside in separate tablespaces and a minor modification to the KS-database interface would facilitate remote database access using SQL*Net[6].

The design assumes each KS will remain centralized on the machine containing the MKS allowing rapid determination of target sites and local data integrity verification. This capability puts the burden of processing on the local machine, thereby minimizing network traffic, impact on remote machines and databases, and probability of remote error.

IAMS design implementation directly supports the three goals of nonintrusive access:

---

[6] Oracle Corporation network protocol tool.

1. The database design does not have to be altered.
2. The access is query only, allowing no updates.
3. The design minimizes the impact multiple access has on the normal database performance.

This last achievement is due primarily to the fact that query construction is done on the local machine and also that the query construction is based on a prioritization of which columns are most likely to contain target data.

More information is stored in KS tables than is currently utilized. This additional application data provide flexibility for future growth. Oracle also provides a wealth of application information in system tables, which may be looked on as a virtual knowledge source in itself.

Another major advantage of IAMS prototype implementation is the use of standard naming conventions. Strict conformance to standards allows for rapid and automated integration of a database in IAMS. As previously stated, if standards were not followed, a synonym table must be constructed, or the database names modified to standards. The latter option is normally cost prohibitive and does not meet nonintrusive integration requirements, because once the names are changed, all database applications must be modified to reflect the changes.

4.9 Future Work

Performance is an issue that was considered during the design, but not strongly emphasized during prototype implementation. A modification to allow for parallel KS processing is needed for full implementation of multiple large

databases. The KS process is a separate executable which accepts the database name and other input. This design facilitates the needed modification.

Post query processing implementation is needed to provide full co-relational capability. More intelligence is necessary to provide the means to automatically construct a query based on the results of another.

In any database design there exists the potential for data dependent database structures. While it is scientifically desirable from a conceptual modelling viewpoint to prevent structural dependencies, it is not always practical. Even though the prototype databases chosen were deliberately small, one such data dependency exists.

In the PERSONNEL database, the primary key of the PERSONNEL entity is SSN. The INSTRUCTOR entity is a subset of the PERSONNEL entity, therefore SSN in INSTRUCTOR should be a foreign key. The automated KS construction routine does not have the intelligence to determine if instructor is a subset of PERSONNEL or vice versa. Therefore, it records SSN as a primary key for each entity leading to potential anomalies when queries are based on SSN. The preprocessing module can not distinguish priority between INSTRUCTOR.SSN and PERSONNEL.SSN.

The preceding situation illustrates the potential need for a tool to allow an application expert to verify KS findings. An application expert would know these potentially complex relationships, and could adjust the knowledge base accordingly

with a minimum of interaction.

The only queries currently supported in the IAMS prototype are simple queries to one table, or a join between two tables. The tables may be contained in the same application, but they do not have to be. Future expansions might include a means of providing additional query capabilities. The dynamic SQL module is robust, and it will handle any legal SQL query. The query construction module would have to be modified to provide additional intelligence in the construction routine.

5 Conclusions

Although there are issues to resolve concerning automatic integration of existing database application into IAMS, the capability of accessing multiple database through the use of intelligence is not only conceptually feasible but highly desirable. The IAMS prototype implementation was easier to attain partly due to the tight constraints placed on the database applications.

Automatic KS generation was made feasible through the use of standard naming conventions. While these conventions are desired for future systems, they are not always strictly interpreted and completely enforced across separate applications. Most existing database applications were built before the standards were published. Therefore, incorporating existing systems into an IAMS network (original IAMS goal) is not as automatic as it becomes using standard names.

Another analysis process is necessary to construct a synonym

table for nonstandard systems which maps or cross references relationships among database applications. Modification to the KS generation utilization processes is also necessary to use the cross reference.

The automated construction of the KS and MKS also revealed an unexpected benefit of standard naming conventions. The conventions facilitate automated data integrity checking. The KS construction process determines if there are conflicts in data type and size for a data element, and the MKS construction process determines any conflicts in data type and size among multiple databases. This discovery makes apparent one of many possible methods of automated data integrity checking among databases built using the standard naming conventions.

Michael Brodie speaks of "interconnectivity" as being the key to future expansion of existing databases (Brodie, 1988). There is currently much research in this area, and the research in IAMS and other recent literature, is convincing proof that the industry is motivated in this investigation. Networks of inter-related databases are being attempted in industry and government today. Automated integration of existing databases, as illustrated by IAMS, provides a significant contribution to this endeavor.

6 References

Bright, M. W., Hurson, A. R., and Pakzad, Simin H. 1992. "A Taxonomy and Current Issues in Multidatabase Systems," IEEE Computer, March, 1992.

Brodie, Michael L. 1988. "Future Intelligent Information Systems: AI and Database Technologies Working Together," Readings in Artificial Intelligence and Databases, John Mylopolous and Michael Brodie, ed., Morgan Kaufmann Publishers, Inc., San Mateo, CA, pp 623-641.

Charniak, E., and McDermott, D. 1986. Introduction to Artificial Intelligence, Addison-Wesley, Reading, MA, pp 1-11.

DeYoung, Tice. 1984. "Voice Interactive Systems Technology (VIST) Research," Interim Technical Report ETL-0349, U.S. Army Corps of Engineers, Engineer Topographic Laboratories, Fort Belvoir, VA.

Dilts, D. W., and Wu, W. 1991 (Jun). "Using Knowledge-Based Technology to Integrate CIM Databases," IEEE Transactions on Knowledge and Engineering, Vol 3, No. 2. pp 237-245.

Dunn, J. E., 1990. "Advanced Studies of Target Systems Suitable for Evaluation of a Hit-to-Kill Terminal Interceptor at White Sands Missile Range," Technical Report DNA-TR-89-235, Defense Nuclear Agency, Alexandria, VA.

Elmasri, R., Navathe, S. B. 1989. Fundamentals of Database Systems, The Benjamin/Cummings Publishing Company, Inc., Redwood City, CA., pp 175-176.

Headquarters, Department of the Army. 1989 (Sep). "Army Data Management and Standards Program," Army Regulation AR 25-9, Washington, DC.

Kaula, R. 1990. An Open Intelligent Information Systems Architecture, Ph.D. Dissertation, UMI Dissertation Information Service, Ann Arbor, MI.

Maune, D. F. 1991 (May-Jun). "Topographic Engineer Technology...Vital in Desert Storm," ARMY RD&A BULLETIN, PB 70-91-3, pp 22-25.

Meiran, Harvey B. 1988. "Use of Mobile Robots in Responding to Radiological and Toxic Chemical Accidents," Proceedings of the Conference on Space and Military Applications of Automation and Robotics, U.S. Army Missile Command, and the National Aeronautics and Space Administration, Marshall Space Flight Center, Huntsville, AL, pp 263-273.

Nii, H. P. 1986. "Blackboard Systems: The Blackboard Model of Problem Solving and the Evolution of Blackboard Architecture," The AI Magazine, Vol VII, No. 2, pp 38-53.

Özsu, M. Tamer, and Valduriez, Patrick. 1992. "Distributed Database Systems: Where Are We Now?," Database Programming & Design, March 1992.

Parsaye, K., Chignell, M., Khoshafian, S., and Wong, H. 1989. Intelligent Databases, Wiley, New York, NY.

Pro*C User's Guide, Version 1.1. 1990. Oracle Corporation.

Swaby, Peter Alan. 1992. "VIDES: An Expert system for Visually Identifying Microfossils," IEEE Expert, Vol 7, No. 2, pp 36-42.

Wright, Peggy. 1992. "Potential Methodologies of Intelligent Access to Multiple Databases," Miscellaneous Paper ITL-92-1, Department of the Army, Waterways Experiment Station, Corps of Engineers, Vicksburg, MS.

APPENDIX A

DATABASE DESIGN

Appendix A Database Design

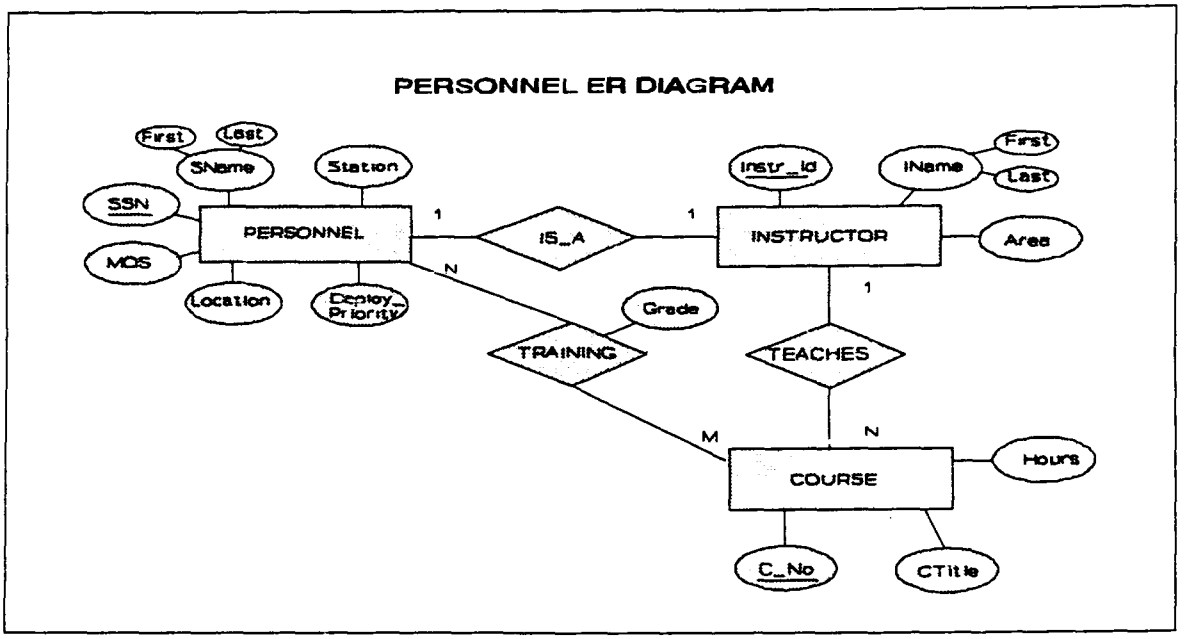


FIGURE A-1 PERSONNEL ENTITY RELATIONSHIP DIAGRAM

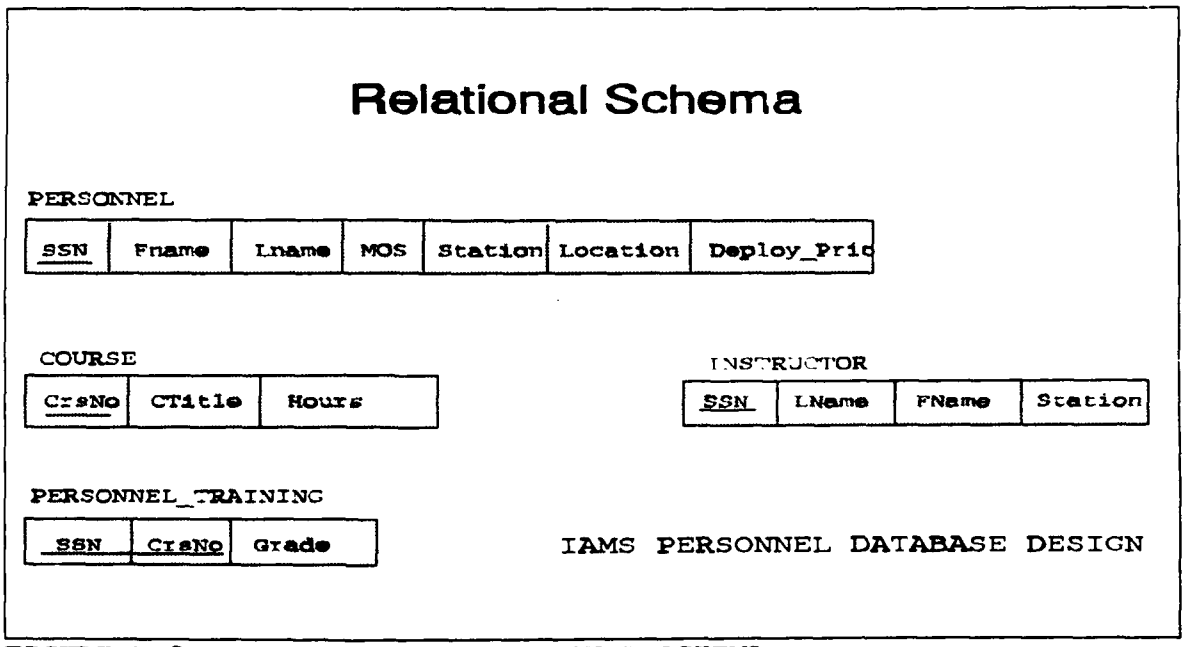


FIGURE A-2 PERSONNEL RELATIONAL SCHEMA

Appendix A Database Design

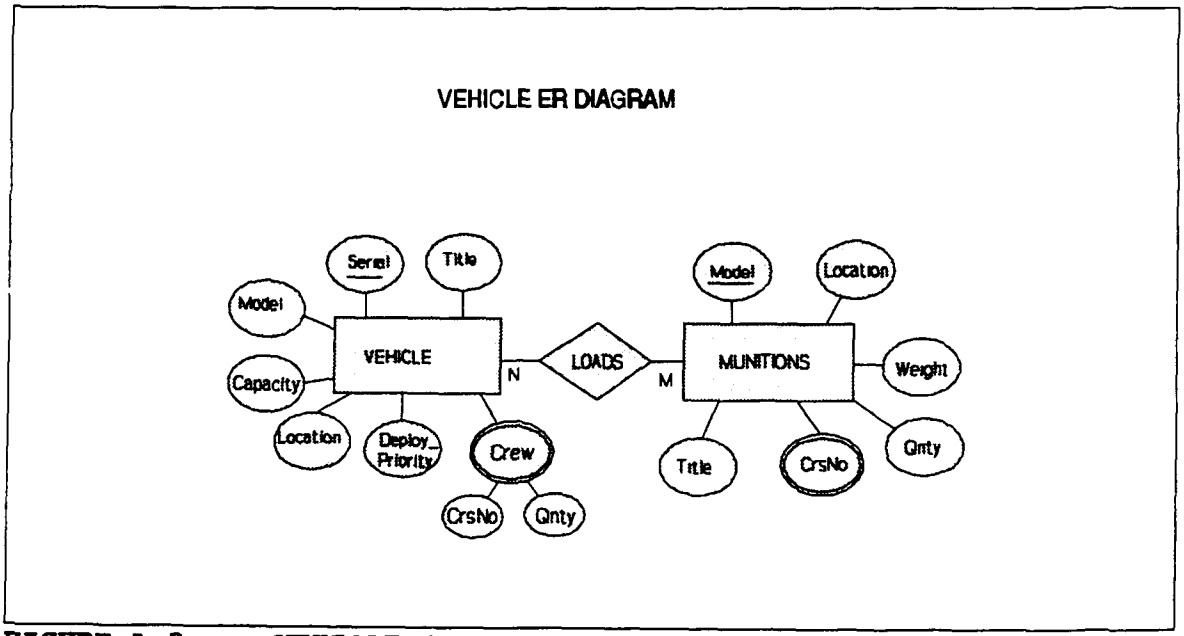


FIGURE A-3 VEHICLE ENTITY RELATIONSHIP DIAGRAM

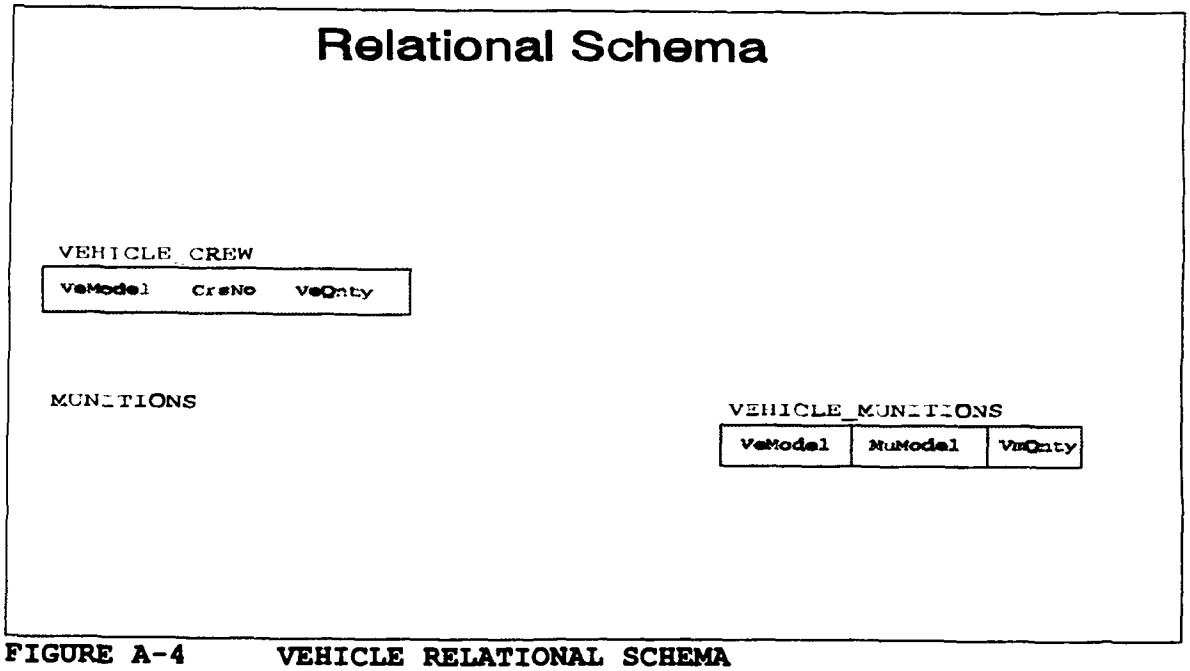


FIGURE A-4 VEHICLE RELATIONAL SCHEMA

APPENDIX B

SQL CODE

Appendix B SQL Script Files

IAMS SQL script files

The purpose of this table is to document the SQL scripts used to extract database application data for Knowledge Source construction. The SQL scripts are completely generic, they can be used against any ORACLE application by passing in the tablespace name. The batch script used to invoke the analysis process is edited to include the ORACLE tablespace, userid and password that has access to the application tablespace. Output files have the file extension *.lst*, and are used to verify the results of the SQL scripts. All scripts are included in this appendix, and example lst files are included when the output is meaningful.

FILE NAME	PURPOSE	OUTPUT
dbrpt.sql	SQL script that first drops any existing IAMS tables, and then starts each script in succession. Input is the application tablespace name.	none
info.sql	SQL script that creates table IAMS_TAB_NAMES, which contains the application table names.	none
info1.sql	SQL script that generates listings of table definitions, columns in a table, tables by column, table indexes, and indexes by column name. Input is tablespace name.	db1a.lst db1b.lst db1c.lst db1d.lst
info2.sql	SQL script that creates table IAMS_TAB_INFO, which contains column info such as data type, data length, data precision and whether or not the column is allowed to be NULL. Output is used to check successful run of script only.	db2.lst
info3.sql	SQL script that creates table IAMS_IND_INFO, which contains index info such as whether the key is primary, foreign, composite, and the columns comprising the key. Output is used to check successful run of script only.	db3.lst
info4.sql	SQL script that generates index report, containing info on key classification. Input is tablespace name.	db4.lst
info5.sql	SQL script that creates table IAMS_FK_STATS which contains information about foreign key statistics.	db5.lst
info6.sql	SQL script that generates a report on foreign key statistics. Input is tablespace name.	db6.lst
info7.sql	SQL script generates report containing index statistics. Input is tablespace name.	db7.lst
mks.sql	SQL script that starts meta knowledge scripts in succession.	none
mks1.sql	SQL script that creates table MKS_TAB_NAMES that contains table names from all databases.	mks1.lst
mks2.sql	SQL script that creates table MKS_TAB_INFO that contains columns from all databases.	mks2.lst

Appendix B SQL Script Files

FILE NAME	PURPOSE	OUTPUT
mks3.sql	SQL script that creates table MKS_IND_INFO that contains index information on all databases.	mks3.lst
mks4.sql	SQL script that creates table MKS_FK_STATS that contains foreign key statistics for all databases.	mks4.lst
mks5.sql	SQL script that creates table MKS_RELATIONS that builds and stores information about relationships among databases.	mks5.lst
mks6.sql	SQL script that generates a report on foreign key statistics.	mks6.lst
mks7.sql	SQL script that generates a report on index statistics.	mks7.lst

Appendix B SQL Script Files

dbrpt.sql Page 1

```
/*      JUL 92          P Wright    */
/*      script file to kick off database analysis scripts */
/*       this analysis is for IAMS research to determine */
/*       database relationships and build a KS basis      */
/*      input parameter is the database application name  */
set termout off
set echo off
set feedback off
set verify off
drop table iams_fk_stats
/
drop table iams_ind_info
/
drop table iams_tab_info
/
start info '&1';
start info1 '&1';
start info2;
start info3;
start info4 '&1';
start info5;
start info6 '&1';
start info7 '&1';
exit
```

Appendix B SQL Script Files

info.sql Page 1

```
/* set up   must run before other info scripts */
/*      july 92   p wright                     */
/*    save application table names to iams_tab_names */
/*    input parameter is application name          */
set termout off
set echo off
set verify off
drop table iams_tab_names
/
create table iams_tab_names (
            application char(30) NOT NULL,
            table_name  char(30) NOT NULL )
/
create unique index tname on iams_tab_names(table_name)
/
insert into iams_tab_names
      (application,table_name)
            select tablespace_name,table_name
            from   user_tables
            where  tablespace_name = '&1'
            and    table_name     != 'IAMS_TAB_NAMES'
/
```

Appendix B SQL Script Files

infol.sql Page 1

```
/*   JUL 92     P Wright                          */
/*     determine application table defns and     */
/* additional info about columns and indexes */
/*    input parameter is application name         */
set termout off
set echo off
set feedback 0
set verify off
set pages 60
set lin 80
col looktime noprint new_value printtime
col table_name format A20 hea 'TABLE' jus C
col index_name format A10 hea 'INDEX' jus C
col column_name format A20 hea 'COLUMN' jus C
col nullind format A8 hea 'NULL ?' jus C
col dtype format A18 hea 'DATA TYPE' jus C
bre on table_name ski 1
spool dbla
tti left printtime center 'IAMS &1 DB - TABLE DEFINITIONS' -
      right 'PAGE:' format 999 sql.pno ski 3
select to_char(sysdate,'MON"-"DD"-"YY') looktime,
       table_name, column_name,
       data_type||'( '||to_char(data_length,'999')||' )' dtype,
       decode(nullable,'N','NOT NULL','  NULL  ') nullind
from   all_tab_columns
where    table_name in
      (select table_name
       from   iams_tab_names)
order by 2,3
/
spool off
bre on column_name ski 1
spool dblb
tti left printtime center 'IAMS &1 DB - TABLES BY COLUMNS' -
      right 'PAGE:' format 999 sql.pno ski 3
select to_char(sysdate,'MON"-"DD"-"YY') looktime,
       column_name, table_name,
       data_type||'( '||to_char(data_length,'999')||' )' dtype,
       decode(nullable,'N','NOT NULL','  NULL  ') nullind
from   all_tab_columns
where  table_name in
      (select table_name
       from   iams_tab_names)
order by 2,3
/
spool off
bre on table_name ski 2 on index_name ski 1
spool dblc
tti left printtime center 'IAMS &1 DB - TABLE INDEXES ' -
      right 'PAGE:' format 999 sql.pno ski 3
select to_char(sysdate,'MON"-"DD"-"YY') looktime,
       table_name, index_name, column_name
from   all_ind_columns
where   table_name in
           (select table_name
            from   iams_tab_names)
```

Appendix B SQL Script Files

infol.sql Page 2

```
order by 2,3,4
/
spool off
bre on column_name ski 2 on table_name ski 1
spool dbld
tti left printtime center 'IAMS &1 DB- INDEXES BY COLUMNS' -
      right 'PAGE:' format 999 sql.pno ski 3
select to_char(sysdate,'MON"-"DD"-"YY') looktime,
            column_name, table_name, index_name
from   all_ind_columns
where  table_name in
            (select table_name
             from   iams_tab_names)
order by 2,3,4
/
spool off
```

Appendix B SQL Script Files

dbla.lst Page 1

AUG-08-92 IAMS VEHICLE DB - TABLE DEFINITIONS PAGE: 1

TABLE	COLUMN	DATA TYPE	NULL ?
MUNITIONS	LOCATION	CHAR(8)	NOT NULL
	MUMODEL	CHAR(6)	NOT NULL
	MUQNTY	NUMBER(22)	NOT NULL
	MUTITLE	CHAR(15)	NULL
	MUWGT	NUMBER(22)	NULL
MUNITIONS_TRAINING	CRSNO	CHAR(6)	NOT NULL
	MUMODEL	CHAR(6)	NOT NULL
STATIC_VEHICLE	VEMANU	CHAR(30)	NOT NULL
	VEMODEL	CHAR(6)	NOT NULL
	VETITLE	CHAR(15)	NOT NULL
VEHICLE	DEPLOY_PRIO	NUMBER(22)	NULL
	LOCATION	CHAR(8)	NOT NULL
	VECAP	NUMBER(22)	NOT NULL
	VEMODEL	CHAR(6)	NOT NULL
	VESERIAL	CHAR(6)	NOT NULL
	VETITLE	CHAR(15)	NULL
VEHICLE_CREW	CRSNO	CHAR(6)	NOT NULL
	VEMODEL	CHAR(6)	NOT NULL
	VEQNTY	NUMBER(22)	NOT NULL
VEHICLE_MUNITIONS	MUMODEL	CHAR(6)	NOT NULL
	VEMODEL	CHAR(6)	NOT NULL
	VMQNTY	NUMBER(22)	NOT NULL

Appendix B SQL Script Files

dblb.lst Page 1

AUG-08-92 IAMS VEHICLE DB - TABLES BY COLUMNS PAGE: 1

COLUMN	TABLE	DATA TYPE	NULL ?
CRSNO	MUNITIONS_TRAINING	CHAR(6)	NOT NULL
	VEHICLE_CREW	CHAR(6)	NOT NULL
DEPLOY_PRIO	VEHICLE	NUMBER(22)	NULL
LOCATION	MUNITIONS	CHAR(8)	NOT NULL
	VEHICLE	CHAR(8)	NOT NULL
MUMODEL	MUNITIONS	CHAR(6)	NOT NULL
	MUNITIONS_TRAINING	CHAR(6)	NOT NULL
	VEHICLE_MUNITIONS	CHAR(6)	NOT NULL
MUQNTY	MUNITIONS	NUMBER(22)	NOT NULL
MUTITLE	MUNITIONS	CHAR(15)	NULL
MUWGT	MUNITIONS	NUMBER(22)	NULL
VECAP	VEHICLE	NUMBER(22)	NOT NULL
VEMANU	STATIC_VEHICLE	CHAR(30)	NOT NULL
VEMODEL	STATIC_VEHICLE	CHAR(6)	NOT NULL
	VEHICLE	CHAR(6)	NOT NULL
	VEHICLE_CREW	CHAR(6)	NOT NULL
	VEHICLE_MUNITIONS	CHAR(6)	NOT NULL
VEQNTY	VEHICLE_CREW	NUMBER(22)	NOT NULL
VESERIAL	VEHICLE	CHAR(6)	NOT NULL
VETITLE	STATIC_VEHICLE	CHAR(15)	NOT NULL
	VEHICLE	CHAR(15)	NULL
VMQNTY	VEHICLE_MUNITIONS	NUMBER(22)	NOT NULL

Appendix B SQL Script Files

dblc.lst Page 1

AUG-08-92 IAMS VEHICLE DB - TABLE INDEXES PAGE: 1

TABLE	INDEX	COLUMN
MUNITIONS	MUN	MUMODEL
MUNITIONS_TRAINING	MUTRN	CRSNO MUMODEL
STATIC_VEHICLE	STVH	VEMODEL
VEHICLE	VEH	VESERIAL
	VHMOD	VEMODEL
VEHICLE_CREW	VCRS	CRSNO VEMODEL
VEHICLE_MUNITIONS	VECAP	MUMODEL VEMODEL

Appendix B SQL Script Files

dbld.lst Page 1

AUG-08-92 IAMS VEHICLE DB- INDEXES BY COLUMNS PAGE: 1

COLUMN	TABLE	INDEX
CRSNO	MUNITIONS_TRAINING	MUTRN
	VEHICLE_CREW	VCRS
MUMODEL	MUNITIONS	MUN
	MUNITIONS_TRAINING	MUTRN
	VEHICLE_MUNITIONS	VECAP
VEMODEL	STATIC_VEHICLE	STVH
	VEHICLE	VHMOD
	VEHICLE_CREW	VCRS
	VEHICLE_MUNITIONS	VECAP
VESERIAL	VEHICLE	VEH

Appendix B SQL Script Files

info2.sql Page 1

```
/*    JUL 92          P Wright  */
/*    save table names & column names */
spool db2
drop table IAMS_TAB_INFO
/
create table IAMS_TAB_INFO as
select table_name, column_name, data_type, data_length,
      data_precision, nullable
from   all_tab_columns atc
where  atc.table_name in
            (select table_name
             from   iams_tab_names)
/
commit;
drop index IAMS_TAB_INFO_INDX
/
create unique index IAMS_TAB_INFO_INDX on IAMS_TAB_INFO
      (column_name, table_name)
/
commit;
spool off
```

Appendix B SQL Script Files

info3.sql Page 1

```
/*   JUL 92                P Wright */
/*   capture key relationships in a database application */
set termout off
set echo off
spool db3
drop table iams_ind_info
/
create table iams_ind_info
        (table_name     char(30),
         index_name     char(30),
         primary        char,
         foreign        char,
         composite      char,
         uniqueness     char,
         column_name    char(30))
/
insert into iams_ind_info
(table_name, index_name, column_name)
select table_name, index_name, column_name
from   all_ind_columns
where  table_name in
            (select table_name
             from iams_tab_names)
/
commit;
update iams_ind_info
set    primary = 'Y',uniqueness = 'Y'
where  index_name in
      (select index_name
       from user_indexes a
       where uniqueness = 'UNIQUE')
and    table_name in
      (select table_name
       from iams_tab_names)
/
commit;
update iams_ind_info
set    composite = 'N'
where  index_name in
      (select index_name
       from user_indexes a
       where 1 =
            (select count(index_name)
             from   all_ind_columns b
             where   a.index_name = b.index_name) )
and    table_name in
      (select table_name
       from   iams_tab_names)
/
commit;
update iams_ind_info
set    foreign = 'N'
where  primary = 'Y'
and    composite = 'N'
/
commit;
```

info3.sql Page 2

```
update iams_ind_info
set    primary = 'N',uniqueness = 'N'
where  primary is NULL
/
commit;
update iams_ind_info
set    composite = 'Y'
where  composite is NULL
/
commit;
update iams_ind_info
set foreign = 'Y'
where  column_name in
      (select a.column_name
       from   iams_ind_info a, iams_ind_info b
       where a.foreign is NULL
       and a.column_name = b.column_name
       and a.table_name <> b.table_name
       and b.primary = 'Y'
       and b.foreign = 'N')
and foreign is NULL
/
commit;
update iams_ind_info
set    foreign = 'N'
where  foreign is NULL
/
commit;
create index IAMS_IND_INFO_IND_1 on iams_ind_info(index_name)
/
commit;
spool off
```

Appendix B SQL Script Files

info4.sql Page 1

```
/*   JUL 92     P Wright                          */
/*   reports on a database application            */
/*  input parameter is the application name */
set termout off
set echo off
set feedback 0
set verify off
set pages 60
set lines 80
col table_name format a20 hea 'TABLE' jus C
col column_name format a15 hea 'COLUMN' jus C
col index_name format a8 head 'INDEX' jus C
col primary format a7 hea 'PRIMARY' jus C
col foreign format a7 hea 'FOREIGN' jus C
col composite format a9 hea 'COMPOSITE' jus C
col uniqueness format a6 hea 'UNIQUE' jus C
bre on table_name ski 1
spool db4
tti center 'IAMS &1 DB - KEY STATISTICS' -
      right 'PAGE:' format 999 sql.pno ski 3
select table_name, column_name, index_name,
       primary, foreign, composite, uniqueness
from   iams_ind_info
order by table_name, index_name
/
spool off
/
```

Appendix B SQL Script Files

db4.lst Page 1

IAMS VEHICLE DB - KEY STATISTICS PAGE: 1

TABLE	COLUMN	INDEX	PRIMARY	FOREIGN	COMPOSITE	UNIQUE
MUNITIONS	MUMODEL	MUN	Y	N	N	Y
MUNITIONS_TRAINING	MUMODEL	MUTRN	Y	Y	Y	Y
	CRSNO	MUTRN	Y	N	Y	Y
STATIC_VEHICLE	VEMODEL	STVH	Y	N	N	Y
VEHICLE	VESERIAL	VEH	Y	N	N	Y
	VEMODEL	VHMOD	N	Y	N	N
VEHICLE_CREW	VEMODEL	VCRS	Y	Y	Y	Y
	CRSNO	VCRS	Y	N	Y	Y
VEHICLE_MUNITIONS	VEMODEL	VECAP	Y	Y	Y	Y
	MUMODEL	VECAP	Y	Y	Y	Y

Appendix B SQL Script Files

info5.sql Page 1

```
/* JUL   92   P Wright                    */
/*  capture foreign key stats for a database application */
set termout off
set echo off
spool db5
drop table iams_fk_stats;
commit;
create table iams_fk_stats
        (table_name     char(30),
         nbr_cols       number(5),
         nbr_shared     number(5),
         nbr_fks        number(5))
/
insert into iams_fk_stats
      (table_name, nbr_cols, nbr_shared, nbr_fks)
select table_name, 0, 0, 0
from   iams_tab_names
/
commit;
create index STATS_IND on iams_fk_stats(table_name)
/
commit;
update iams_fk_stats fs
set    fs.nbr_cols =
      (select count(i.table_name)
      from    iams_tab_info i
      where   i.table_name = fs.table_name)
/
commit;
update iams_fk_stats fs
set fs.nbr_shared =
      (select count(i.table_name)
      from    iams_tab_info i
      where   i.table_name = fs.table_name
and exists
      (select 'x'
      from    iams_tab_info i2
      where   i2.column_name = i.column_name
      and     i2.table_name <> i.table_name))
/
commit;
update iams_fk_stats fs
set fs.nbr_fks =
      (select count(i.index_name)
            from   iams_ind_info i
       where  i.table_name = fs.table_name
       and    i.foreign    = 'Y' )
/
commit;
spool off
```

Appendix B SQL Script Files

info6.sql Page 1

```
/*  JUL 92                  P Wright          */
/*  report foreign key statistics             */
/*  input parameter is the application name */
set termout off
set echo off
set feedback 0
set verify off
set pages 60
set lines 80
col table_name format A20 hea 'TABLE' jus C
col nbr_cols hea 'NBR COLS' jus C
col nbr_shared hea 'NBR COMMON COLS' jus C
col nbr_fks hea 'NBR FKS' jus C
bre on table_name ski 1
spool db6
tti center 'IAMS &1 DB - INDEX STATS BY TABLE' -
       right 'PAGE:' format 999 sql.pno ski 3
select table_name, nbr_cols, nbr_shared, nbr_fks
from   iams_fk_stats
order by table_name
/
spool off
```

Appendix B SQL Script Files

db6.lst Page 1

IAMS VEHICLE DB - INDEX STATS BY TABLE PAGE: 1

TABLE	NBR COLS	NBR COMMON COLS	NBR FKS
MUNITIONS	5	2	0
MUNITIONS_TRAINING	2	2	1
STATIC_VEHICLE	3	2	0
VEHICLE	6	3	1
VEHICLE_CREW	3	2	1
VEHICLE_MUNITIONS	3	2	2

Appendix B SQL Script Files

info7.sql Page 1

```
/*      JUL 92       P Wright    */
/*      report on index stats    */
/* input parameter is application name */
set termout off
set echo off
set feedback 0
set verify off
set pages 60
set lines 80
spool db7
col column_name format a20 hea 'COLUMN' jus C
col table_name format a20 hea 'TABLE' jus C
col index_name format a10 hea 'INDEX' jus C
col looktime format a10 hea 'DATE' jus C
col uniqueness format a9 hea 'UNIQUE' jus C
bre on column_name ski 2
bre on table_name ski 1
tti left printtime center 'IAMS &1 DB - INDEXES BY COLUMNS' -
       right 'PAGE:' format 999 sql.pno ski 3
select to_char(sysdate,'MON"-"DD"-"YY') looktime,
      a.column_name, a.table_name, a.index_name, b.uniqueness
from   all_ind_columns a, user_indexes b
where  a.table_name  = b.table_name
and    a.index_owner = b.table_owner
and    a.index_name  = b.index_name
and    a.table_name in
            (select table_name
             from   iams_tab_names)
order by 2,3,4
/
spool off
/
```

Appendix B SQL Script Files

db7.lst Page 1

AUG-08-92 IAMS VEHICLE DB - INDEXES BY COLUMNS PAGE: 1

COLUMN	TABLE	INDEX	UNIQUE
CRSNO	MUNITIONS_TRAINING	MUTRN	UNIQUE
CRSNO	VEHICLE_CREW	VCRS	UNIQUE
MUMODEL	MUNITIONS	MUN	UNIQUE
MUMODEL	MUNITIONS_TRAINING	MUTRN	UNIQUE
MUMODEL	VEHICLE_MUNITIONS	VECAP	UNIQUE
VEMODEL	STATIC_VEHICLE	STVH	UNIQUE
VEMODEL	VEHICLE	VHMOD	NONUNIQUE
VEMODEL	VEHICLE_CREW	VCRS	UNIQUE
VEMODEL	VEHICLE_MUNITIONS	VECAP	UNIQUE
VESERIAL	VEHICLE	VEH	UNIQUE

Appendix B SQL Script Files

Aug 13 17:36 1992 mks.sql Page 1

```
/*      august 92   p wright                    */
/*     script file to kick off database analysis scripts */
/*      this analysis is for IAMS research to determine */
/*      application relationships and build a MKS basis     */
set termout off
set echo off
set feedback off
set verify off
start mks1;
start mks2;
start mks3;
start mks4;
start mks5;
start mks6;
start mks7;
start mks8;
exit
```

Appendix B SQL Script Files

Aug 13 17:24 1992 mks1.sql Page 1

```
/* set up   must run before other info scripts */
/*       july 92   p wright                    */
/*    save application table names to MKS_TAB_NAMES */
set termout off
set echo off
set verify off
spool mks1
drop table MKS_TAB_NAMES
/
create table MKS_TAB_NAMES (
            application char(30) NOT NULL,
            table_name  char(30) NOT NULL )
/
create urique index mkname on MKS_TAB_NAMES(application,table_name)
/
/*    get table names from PERSONNEL application */
insert into MKS_TAB_NAMES
      (application,table_name)
            select application,table_name
            from   p_tnames
/
/*    get table names from VEHICLE application */
insert into MKS_TAB_NAMES
      (application,table_name)
            select application,table_name
            from   v_tnames
/
commit
/
spool off
```

Appendix B SQL Script Files

Aug 13 17:24 1992 mks2.sql Page 1

```
/*    JUL 92          P Wright  */
/*    save table names & column names */
spool mks2
drop table MKS_TAB_INFO
/
create table MKS_TAB_INFO as
select a.application,b.table_name, b.column_name, b.data_type, b.data_length,
      b.data_precision, b.nullable
from   MKS_tab_names a, p_tinf b
where  a.application = 'PERSONNEL'
and    a.table_name  = b.table_name
/
commit
/
insert into MKS_TAB_INFO
select a.application,b.table_name, b.column_name, b.data_type, b.data_length,
      b.data_precision, b.nullable
from   MKS_tab_names a, v_tinf b
where  a.application = 'VEHICLE'
and    a.table_name  = b.table_name
/
drop index MKS_TINF_INDX
/
create unique index MKS_TINF_INDX on MKS_TAB_INFO
      (application, table_name, column_name)
/
commit
/
spool off
```

Appendix B SQL Script Files

Aug 13 17:41 1992 mks3.sql Page 1

```
/*    JUL 92          P Wright  */
/*    save table names & column names */
spool mks3
drop table MKS_IND_INFO
/
create table MKS_IND_INFO as
select a.application,b.table_name, b.index_name, b.primary, b.foreign,
      b.composite, b.uniqueness, b.column_name
from   MKS_tab_names a, p_iinf b
where  a.application = 'PERSONNEL'
and    a.table_name  = b.table_name
/
commit
/
insert into MKS_IND_INFO
select a.application,b.table_name, b.index_name, b.primary, b.foreign,
      b.composite, b.uniqueness, b.column_name
from   MKS_tab_names a, v_iinf b
where  a.application = 'VEHICLE'
and    a.table_name  = b.table_name
/
drop index MKS_IINF_INDX
/
create unique index MKS_IINF_INDX on MKS_IND_INFO
      (application, table_name, column_name )
/
commit
/
spool off
```

Appendix B SQL Script Files

Aug 13 17:42 1992 mks4.sql Page 1

```
/* JUL   92   P Wright                    */
/*  capture foreign key stats for a database application */
set termout off
set echo off
spool mks4
drop table MKS_FK_STATS
/
commit
/
create table MKS_FK_STATS
        (application    char(30),
       table_name     char(30),
         nbr_cols       number(5),
         nbr_shared     number(5),
         nbr_fks        number(5))
/
commit
/
create index MSTATS_IND on MKS_FK_STATS(application,table_name)
/
commit
/
insert into MKS_FK_STATS
      select 'PERSONNEL',table_name,nbr_cols, nbr_shared, nbr_fks
      from   p_fks
/
commit
/
insert into MKS_FK_STATS
      select 'VEHICLE',table_name,nbr_cols, nbr_shared, nbr_fks
      from   v_fks
/
commit
/
spool off
```

Appendix B SQL Script Files

Aug 13 17:19 1992 mks5.sql Page 1

```
/* script file to capture master relations between applications */
/*      july 92   p wright                                      */
set termout off
set echo off
spool mks5
drop table MKS_RELATIONS
/
create table MKS_RELATIONS(
            column_name char(30) NOT NULL,
            super_key   char(1)  ,
            key         char(1)  ,
            nbr_apps    number   ,
            nbr_occurs  number   )
/
drop index mkrel
/
create unique index mkrel on MKS_RELATIONS(column_name)
/
commit
/
insert into MKS_RELATIONS
      (column_name,super_key,key,nbr_apps,nbr_occurs)
            select distinct column_name,NULL,NULL,0,0
            from   MKS_TAB_INFO
/
commit
/
update MKS_RELATIONS
      set key = 'Y'
      where column_name in
            (select distinct column_name
             from   MKS_IND_INFO )
/
commit
/
update MKS_RELATIONS
      set super_key = 'Y'
      where column_name in
            (select distinct a.column_name
             from   MKS_IND_INFO a, MKS_IND_INFO b
             where  a.column_name =  b.column_name
             and    a.application <> b.application )
/
commit
/
update MKS_RELATIONS M
set    nbr_occurs =
      (select NVL(count(column_name),0)
       from   MKS_TAB_INFO a
       where  a.column_name = M.column_name )
/
commit
/
update MKS_RELATIONS M
set    nbr_apps =
      (select NVL(count(distinct a.application),0)
```

```
        from   MKS_TAB_INFO a, MKS_TAB_INFO b
        where  a.column_name = b.column_name
        and    a.application <> b.application
        and    a.column_name = M.column_name
        group by a.column_name )
/
commit
/
spool off
```

Appendix B SQL Script Files

Aug 13 17:44 1992 mks6.sql Page 1

```
/*  JUL 92                    P Wright          */
/*  report foreign key statistics               */
set termout off
set echo off
set feedback 0
set verify off
set pages 60
set lines 80
col application format A12 hea 'APPLICATION' jus C
col table_name format A20 hea 'TABLE' jus C
col nbr_cols hea 'NBR COLS' jus C
col nbr_shared hea 'NBR COMMON COLS' jus C
col nbr_fks hea 'NBR FKS' jus C
bre on table_name ski 1
spool mks6
tti center 'IAMS MKS - INDEX STATS BY TABLE' -
        right 'PAGE:' format 999 sql.pno ski 3
select table_name, nbr_cols, nbr_shared, nbr_fks
from   mks_fk_stats
order by application, table_name
/
spool off
```

Appendix B SQL Script Files

Aug 13 17:49 1992 mks6.lst Page 1

IAMS MKS - INDEX STATS BY TABLE PAGE: 1

TABLE	NBR COLS	NBR COMMON COLS	NBR FKS
COURSE	3	1	0
INSTRUCTOR	4	4	0
PERSONNEL	7	4	0
PERSONNEL_TRAINING	3	2	2
MUNITIONS	5	2	0
MUNITIONS_TRAINING	2	2	1
STATIC_VEHICLE	3	2	0
VEHICLE	6	3	1
VEHICLE_CREW	3	2	1
VEHICLE_MUNITIONS	3	2	2

Appendix B SQL Script Files

Aug 13 17:52 1992 mks7.sql Page 1

```
/*   JUL 92     P Wright                          */
/*   reports on a database application            */
/*  input parameter is the application name */
set termout off
set echo off
set feedback 0
set verify off
set pages 60
set lines 90
col application format a12 hea 'APPLICATION' jus C
col table_name format a18 hea 'TABLE' jus C
col column_name format a10 hea 'COLUMN' jus C
col index_name format a8 head 'INDEX' jus C
col primary format a7 hea 'PRIMARY' jus C
col foreign format a7 hea 'FOREIGN' jus C
col composite format a9 hea 'COMPOSITE' jus C
col uniqueness format a6 hea 'UNIQUE' jus C
bre on table_name ski 1
spool mks7
tti center 'IAMS MKS DB - KEY STATISTICS' -
      right 'PAGE:' format 999 sql.pno ski 3
select application, table_name, column_name, index_name,
       primary, foreign, composite, uniqueness
from   mks_ind_info
order by application, table_name, index_name
/
spool off
/
```

Appendix B SQL Script Files

Aug 13 17:52 1992 mks7.lst Page 1

IAMS MKS DB - KEY STATISTICS PAGE: 1

APPLICATION	TABLE	COLUMN	INDEX	PRIMARY	FOREIGN	COMPOSITE	UNIQUE
PERSONNEL	COURSE	CRSNO	CNO	Y	N	N	Y
PERSONNEL	INSTRUCTOR	SSN	INS	Y	N	N	Y
PERSONNEL	PERSONNEL	SSN	PER	Y	N	N	Y
PERSONNEL	PERSONNEL_TRAINING	SSN	PTR	Y	Y	Y	Y
PERSONNEL		CRSNO	PTR	Y	Y	Y	Y
VEHICLE	MUNITIONS	MUMODEL	MUN	Y	N	N	Y
VEHICLE	MUNITIONS_TRAINING	MUMODEL	MUTRN	Y	Y	Y	Y
VEHICLE		CRSNO	MUTRN	Y	N	Y	Y
VEHICLE	STATIC_VEHICLE	VEMODEL	STVH	Y	N	N	Y
VEHICLE	VEHICLE	VESERIAL	VEH	Y	N	N	Y
VEHICLE		VEMODEL	VHMOD	N	Y	N	N
VEHICLE	VEHICLE_CREW	VEMODEL	VCRS	Y	Y	Y	Y
VEHICLE		CRSNO	VCRS	Y	N	Y	Y
VEHICLE	VEHICLE_MUNITIONS	VEMODEL	VECAP	Y	Y	Y	Y
VEHICLE		MUMODEL	VECAP	Y	Y	Y	Y

APPENDIX C

PRO*C CODE

Appendix C Pro*C Code

IAMS Pro*C Source Code

The purpose of this table is to describe the functionality of the individual Pro*C code modules contained in the IAMS system.

FILE	DESCRIPTION
DSQL.PC	**Dynamic SQL:** DSQL.PC contains the code needed to execute the query generated in QC.C. This was taken mostly from the *Oracle Pro*C User's Guide v1.1* with a few modifications. It executes a query which is stored in a character string, and stores the results in a text file.
KS.PC	**Knowledge Source:** This file contains the function main. It starts by loading the KS from either a previously created data file or from Oracle tables created by the SQL code. If the KS data file does not exist, this process creates and saves it. Then it reads the target column(s) from a query file, and builds and executes a query based on the target column(s) and search constraints. The results are stored to a results file so the interface may display them. Errors are trapped and reported to the interface via a code in the results file.
MKS.PC	**Meta-Knowledge Source:** This file contains the code needed to load the MKS either from a previously created data file or from Oracle tables created by the SQL code. If the MKS data file does not exist, this process creates and saves it.

Appendix C Pro*C Code

Oct 11 17:41 1992 dsql.pc Page 1

```
/*****************************************************************************
                              PROGRAM DSQL.PC
   Code adapted from Oracle Pro*C User's Guide.  Reference Appendix E,
         pp. 228-245.
   Specification designed by:
         Peggy Wright in support of IAMS research
   Coded by:
         Shannon Thornton                  15 AUG 92
*****************************************************************************/
#include <stdio.h>
#include <string.h>

#define MAXLENBHVNAMES  0
#define MAXLENBIVNAMES  0
#define MAXLENSHVNAMES 10
#define MAXLENSIVNAMES  0
#define MAXNUMBV       10
#define MAXNUMSV       10

/* SQL codes.  Will be used for bitwise operations so should be a power of 2 */
#define SQL_OK      0              /* Binary -  0000                          */
#define SQL_WARNING 1              /* Binary -  0001                          */
#define SQL_ERROR   2              /* Binary -  0010                          */
#define NOT_FOUND   1403           /* Returned when FETCH returns nothing     */

/* Used for results from process_sql() - True if OK, False if ERROR           */
#define ERROR  -1
#define OK      1

#define min(a, b)   ((a < b) ? (a) : (b))
#define max(a, b)   ((a > b) ? (a) : (b))

extern FILE *iams_log;
extern FILE *iams_res;
extern char app_name[31];

EXEC SQL BEGIN DECLARE SECTION;
   VARCHAR uid[20];
   VARCHAR pwd[20];
   char stmt[4096];
EXEC SQL END DECLARE SECTION;

EXEC SQL INCLUDE SQLCA;
EXEC SQL INCLUDE SQLDA;

SQLDA *bind_da = (SQLDA *) NULL;
SQLDA *select_da = (SQLDA *) NULL;
short *sdt = 0;
int    sdt1;

int check_warning()
{
   if (sqlca.sqlwarn[0] == 'W')
      return SQL_WARNING;
   else return SQL_OK;
}
```

```
int check_error()
{
   switch (sqlca.sqlcode) {
      case SQL_OK:
         return SQL_OK;
      case NOT_FOUND:
         return NOT_FOUND;
      default:
         return SQL_ERROR;
   }
}

sql_warning()
{
   if (sqlca.sqlwarn[1] == 'W')
      fprintf(iams_log, "%s:  SQLWARNING:  Column was truncated.\n",
         app_name);
   else if (sqlca.sqlwarn[2] == 'W') {
      fprintf(iams_log, "%s:  SQLWARNING:  NULL values in aggregate ",
         app_name);
      fprintf(iams_log, "(MAX, SUM) function.\n");
   }
   else if (sqlca.sqlwarn[3] == 'W') {
      fprintf(iams_log, "%s:  SQLWARNING:  INTO var count not equal ",
         app_name);
      fprintf(iams_log, "column count.\n");
   }
   else if (sqlca.sqlwarn[4] == 'W') {
      fprintf(iams_log, "%s:  SQLWARNING:  Update or Delete without ",
         app_name);
      fprintf(iams_log, "WHERE clause.\n");
   }
   else if (sqlca.sqlwarn[5] == 'W')
      fprintf(iams_log, "%s:  SQLWARNING:  ???\n", app_name);
   else if (sqlca.sqlwarn[6] == 'W')
      fprintf(iams_log, "%s:  SQLWARNING:  Rollback required.\n",
         app_name);
   else if (sqlca.sqlwarn[7] == 'W')  {
      fprintf(iams_log, "%s:  SQLWARNING:  Change after query ",
         app_name);
      fprintf(iams_log, "start on Select For Update.\n");
   }
}

print_colnames(dp)
SQLDA *dp;
{
   int len, i, j;
   char name[31];

   fprintf(iams_log,"\n");                          /* LF in the lof file     */
   for (i = 0; i < dp->N; i++) {                    /* Loop for each column   */
      len = min(dp->L[i], dp->C[i]);                /* column length = min of */
      len = min(len, sizeof(name)-1);               /* column length, buffer  */
                                                    /* length, space allocated*/
```

```
                                                        /* for name                */
      memcpy(name, dp->S[i], len);                      /* Copy the column name    */
      name[len] = '\0';                                 /* Terminate string        */
      if (sdt[i] == 2) {                                /* Is it a numeric column?*/
         fprintf(iams_log,"%*s ", dp->L[i], name);      /* Print right justified  */
         fprintf(iams_res,"%*s ", dp->L[i], name);
      }
      else {
         fprintf(iams_log,"%-*s ", dp->L[i],            /* Print left justified   */
            name);
         fprintf(iams_res,"%-*s ", dp->L[i],            /* Print left justified   */
            name);
      }
   }
   fprintf(iams_log, "\n");                             /* Send to the log file   */
   fprintf(iams_res, "\n");                             /* Send to the log file   */
   for (i = 0; i < dp->N; i++) {                        /* Loop for each column   */
      for (j = 0; j < dp->L[i]; j++) {                  /* Loop for column length */
         fprintf(iams_log, "-");                        /* Underline the column   */
         fprintf(iams_res, "-");
      }
      fprintf(iams_log, " ");                           /* One space between cols */
      fprintf(iams_res, " ");
   }
   fprintf(iams_log, "\n");                             /* Send to the log file   */
   fprintf(iams_res, "\n");
}

print_row(dp)
SQLDA *dp;
{
   int i;

   for (i = 0; i < dp->N; i++)                          /* Loop for each column   */
      if (*(dp->I[i]) < 0) {                            /* Returned NULL?         */
         fprintf(iams_log,"%-*s ",dp->L[i], " ");       /* Print nothing          */
         fprintf(iams_res,"%-*s ",dp->L[i], " ");
      }
      else {
         fprintf(iams_log,"%-.*s ",dp->L[i],            /* Print column contents  */
            dp->V[i]);
         fprintf(iams_res,"%-.*s ",dp->L[i],
            dp->V[i]);
      }
   fprintf(iams_log, "\n");                             /* Send to the log file   */
   fprintf(iams_res, "\n");
}

cleanup()
{
   int i;

   if (select_da != NULL) {
      for (i = 0; i < select_da->N; i++) {
         free(select_da->V[i]);
         free(select_da->I[i]);
```

```
        }
        sqlclu(select_da);
        select_da = (SQLDA *) NULL;
    }
    if (bind_da != NULL) {
        sqlclu(bind_da);
        bind_da = (SQLDA *) NULL;
    }
}

init_select_vars(sdp)
SQLDA *sdp;
{
    int i;
    unsigned char prec;
    char scale;

    if (sdt == NULL)                                    /* Allocate sdt 1st time  */
        sdt = (short *) malloc(sizeof(short) * sdp->N);
    else                                                /* Re-allocate sdt        */
        sdt = (short *) realloc(sdt,sizeof(short)*sdp->N);

    for (i = 0; i < sdp->N; i++) {
        sdp->T[i] = (sdp->T[i] & 0x00FF);               /* Clear high order bits  */
        sdt[i] = sdp->T[i];                             /* Store the type         */
        if (sdp->T[i] == 2) {                           /* Is it a numeric field? */
            prec = (unsigned char) (sdp->L[i] >> 8);    /* Get high order byte    */
            scale = (char) sdp->L[i];                   /* Get low order byte     */
            if (prec == 0)                              /* No precision?          */
                prec = 8;                               /* Default to 8           */
                                                        /* THIS WAS 26!!!!!       */
            sdp->L[i] = prec;                           /* Store precision        */
            if (scale < 0)                              /* Negative scale?        */
                sdp->L[i] += -scale;                    /* Must add trailing zeros*/
            sdp->L[i] += 2;                             /* Include possible sign  */
        }                                               /* and decimal point      */
        else if (sdp->T[i] == 12)                       /* Is it a date field?    */
            sdp->L[i] = 9;                              /* DD-MON-YY              */
        sdp->T[i] = 1;                                  /* Coerce to CHAR         */
        sdp->L[i] = min(sdp->L[i], 240);                /* Max len. is 240        */
        sdp->V[i] = (char *) malloc(sdp->L[i]);         /* Allocate room for data */
        sdp->I[i] = (short *) malloc(sizeof(short));    /* Allocate room for ptr  */
    }
}

int process_sql(query)
char *query;
{
    int bdSize = MAXNUMBV, sdSize = MAXNUMSV, rowid;

/*---------------------------------------------------------------------------*/
/*----------------* Copy the query to a host variables  *--------------------*/
/*---------------------------------------------------------------------------*/
    strcpy(stmt, query);
/*---------------------------------------------------------------------------*/
/*--------------------* Check for bogus statements *-------------------------*/
```

Oct 11 17:41 1992 dsql.pc Page 5

```
/*-------------------------------------------------------------------------*/
   if (stmt[strlen(stmt)-1] == ';')
      stmt[strlen(stmt)-1] = '\0';
   if (strlen(stmt) == 0)
      return ERROR;
/*-------------------------------------------------------------------------*/
/*--------------------* Process the SQL statement *--------------------------*/
/*-------------------------------------------------------------------------*/
   EXEC SQL PREPARE S FROM :stmt;
   if ((check_warning() || check_error()) == SQL_ERROR)
      return ERROR;

   EXEC SQL DECLARE C CURSOR FOR S;
   if (check_error() == SQL_ERROR)
      return ERROR;

   bind_da = (SQLDA *) sqlald(bdSize,                    /* Get space for bind vars*/
      MAXLENBHVNAMES, MAXLENBIVNAMES);
   if (bind_da == NULL) {                                /* Error during alloc?    */
      fprintf(iams_log, "Error allocating memory for bind variables.\n");
      exit(ERROR);
   }
   bind_da->N = bdSize;                                  /* Set number of entries  */
   EXEC SQL DESCRIBE BIND VARIABLES FOR S INTO bind_da;
   if (check_error() == SQL_ERROR)                       /* Error during describe? */
      return ERROR;

   if (bind_da->F < 0) {                                 /* Was previous size okay?*/
      bdSize = -(bind_da->F);                            /* Get the correct size   */
      sqlclu(bind_da);                                   /* Free old desc. area    */
      bind_da = (SQLDA *) sqlald(bdSize,                 /* Allocate new space for */
         MAXLENBHVNAMES, MAXLENBIVNAMES);                /* bind variables         */
      if (bind_da == NULL) {                             /* Error during alloc?    */
         fprintf(iams_log, "Error allocating memory for bind variables.\n");
         exit(ERROR);
      }
      EXEC SQL DESCRIBE BIND VARIABLES FOR S INTO bind_da;
      if (check_error() == SQL_ERROR)                    /* Error during describe? */
         return ERROR;
   }
   bind_da->N = bind_da->F;                              /* # entries = # variables*/

   EXEC SQL OPEN C USING DESCRIPTOR bind_da;
   if (check_error() == SQL_ERROR) {                     /* Error during open?     */
      cleanup();                                         /* Deallocate space       */
      return ERROR;
   }

   select_da = (SQLDA *) sqlald(sdSize,                  /* Allocate new space for */
      MAXLENSHVNAMES, MAXLENSIVNAMES);                   /* define variables       */
   if (select_da == NULL) {
      fprintf(iams_log, "Error allocating memory for select variables.\n");
      cleanup();                                         /* Deallocate space       */
      exit(ERROR);                                       /* Close the open cursor  */
   }
   select_da->N = 0;                                     /* Init. # of entries     */
```

```
   EXEC SQL DESCRIBE SELECT LIST FOR S INTO select_da;
   if (check_error() == SQL_ERROR) {                    /* Error during describe? */
      cleanup();                                        /* Deallocate space       */
      EXEC SQL CLOSE C;                                 /* Close the open cursor  */
      return ERROR;
   }

   select_da->N = sdSize;                               /* Set number of entries  */
   if (select_da->F < 0) {                              /* Was previous size OK?  */
      sdSize = -(select_da->F);                         /* Get correct size       */
      sqlclu(select_da);                                /* Free old desc. area    */
      select_da = (SQLDA *) sqlald(sdSize,              /* Get correct amount of  */
         MAXLENSHVNAMES, MAXLENSIVNAMES);               /* space for select vars  */
      if (select_da == NULL) {                          /* Error allocating?      */
         fprintf(iams_log, "Error allocating memory for define variables.\n");
         cleanup();                                     /* Deallocate space       */
         exit(ERROR);
      }
      EXEC SQL DESCRIBE SELECT LIST FOR S INTO select_da;
      if (check_error() == SQL_ERROR) {                 /* Error during describe? */
         cleanup();                                     /* Deallocate space       */
         EXEC SQL CLOSE C;                              /* Close the open cursor  */
         return ERROR;
      }
   }
   select_da->N = select_da->F;                         /* # entries = # variables*/

   if (select_da->F != 0)                               /* # variables <> zero?   */
      init_select_vars(select_da);                      /* Initialize the vars    */
   print_colnames(select_da);                           /* Print column names     */
   for (rowid = 0; ; rowid++) {                         /* Keep count of # of rows*/
      EXEC SQL FETCH C USING DESCRIPTOR select_da;      /* Get a row of results   */
      if (check_error() == SQL_ERROR) {                 /* Error during fetch?    */
         cleanup();                                     /* Deallocate space       */
         EXEC SQL CLOSE C;                              /* Close the open cursor  */
         return ERROR;
      }
      else if (check_error() == NOT_FOUND)              /* No rows left?          */
         break;                                         /* Break out of loop      */
      print_row(select_da);                             /* Print the current row  */
   }
   fprintf(iams_log,"\n%u rows selected.\n\n",          /* Send to the log file   */
      rowid);
   cleanup();                                           /* Deallocate space       */
   EXEC SQL CLOSE C;                                    /* Close the open cursor  */
   return rowid;
}
```

Appendix C Pro*C Code

Oct 11 17:41 1992 ks.pc Page 1

```
/*****************************************************************************
                                    PROGRAM KS.PC
   Project specifications developed by:
         Peggy Wright in support of IAMS research
   Code written by:
         Shannon Thornton           12 AUG 92

   Prompts for userid and password.  Once entered, the program attempts to
   connect to Oracle and create a Knowledge Source based on the IAMS tables.
   The Knowledge Source contains information about the database from three
   different perspectives:  column, table, and index.

   The column perspective contains information about all columns in the
   database.  The column's name, type, size, and number of decimals (if
   numeric) are contained.  Any tables and indexes in which the column is
   used is also stored.

   The table perspective contains information about all tables in the
   database.  The table's name is stored along with any columns and indexes
   used in that table.  Information about primary, foreign, and composite
   keys is stored with each column.  Whether the column can be Null or not
   is also stored.  The uniqueness of an index is stored with each index
   name.

   The index perspective contains information about all index in the
   database.  The index's name, uniqueness, table name where the index is
   used, and all columns forming the index are stored.

                              Naming Convention
   Functions used for manipulating the tree or list structure begin with the
   first letter of the structure used.  For example, to insert a column into
   the column tree, c_ins(..) is called.  The c_ indicates the function uses
   the column tree structure.  All functions that manipulate the table list
   inside a node in the column tree begin with c_t.  Any function that
   manipulates the index list inside the previously mentioned table list
   begins with c_i.  All table tree functions start with t_, and all index
   tree functions start with i_.  This convention is also used for global
   tree variables, c_tree, t_tree, and i_tree, which reference the column
   tree, table tree, and index tree, respectively.
*****************************************************************************/

#include <stdio.h>
#include <string.h>
#include "types.h"

extern struct c_node *c_tree;       /* Pointer to the column tree          */
extern struct t_node *t_tree;       /* Pointer to the table tree           */
extern struct i_node *i_tree;       /* Pointer to the index tree           */

FILE *fd;                           /* File pointer for KS data file       */
FILE *iams_log;                     /* File pointer to the IAMS log file   */
FILE *iams_qry;                     /* File pointer to the IAMS query file */
FILE *iams_res;                     /* File pointer to the query results file */
char app_name[NAMELENGTH];          /* Application name                    */

EXEC SQL BEGIN DECLARE SECTION;
```

Oct 11 17:41 1992 ks.pc Page 2

```
    VARCHAR userid[20];                /* User id used to connect to Oracle      */
   VARCHAR password[20];              /* Password used to connect to Oracle     */
/* - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - */
/* The following host variables are used for storing data returned from       */
/* embedded SQL Queries.                                                      */
/* - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - */
   VARCHAR tbl_name[31];
   VARCHAR col_name[31];
   VARCHAR idx_name[31];
   char nullable;
   char primary;
   char foreign;
   char composite;
   char uniqueness;
   int decimal;
   int size;
   VARCHAR type[10];
EXEC SQL END DECLARE SECTION;
EXEC SQL INCLUDE SQLCA;


prompt()
{
   printf("Need to access %s.\n", app_name);
   printf("\n   Enter User ID:  ");
   scanf("%s",userid.arr);
   userid.len = strlen(userid.arr);
   printf("\n  Enter Password:  ");
   scanf("%s",password.arr);
   password.len = strlen(password.arr);
}

int oracle_login()
/*****************************************************************************
DESCRIPTION:      Prompts for a userid and password.  Once
                  this is entered, it tries to connect to
                  Oracle using the userid and password.
PARAMETERS:       None
RETURNS:          Oracle error code (0 if no error)
CALLS:            Oracle
CALLED BY:        main
*****************************************************************************/
{
   EXEC SQL CONNECT :userid          /* Connect to Oracle using the entered    */
   IDENTIFIED BY :password;          /* user ID and password                   */
   return sqlca.sqlcode;             /* Return the Oracle error code           */
}

init_trees()
/*****************************************************************************
DESCRIPTION:      Initializes the table, column, and index
                  trees.
PARAMETERS:       None
RETURNS:          None
CALLS:            t_init, c_init, i_init
CALLED BY:        main
```

```
*****************************************************************************/
{
   t_init();                          /* Initialize the table tree, t_tree      */
   c_init();                          /* Initialize the column tree, c_tree     */
   i_init();                          /* Initialize the index tree, i_tree      */
}

del_trees()
/*****************************************************************************
DESCRIPTION:       Deletes the table, column, and index
                   trees.
PARAMETERS:        None
RETURNS:           None
CALLS:             t_deltree, c_deltree, i_deltree
CALLED BY:         main
*****************************************************************************/
{
   t_deltree(t_tree);                 /* Delete the table tree                  */
   c_deltree(c_tree);                 /* Delete the column tree                 */
   i_deltree(i_tree);                 /* Delete the index tree                  */
}

get_table_info()
/*****************************************************************************
DESCRIPTION:       Loads table information about a database
                   into a tree structure ordered by
                   table_name.
PARAMETERS:        None
RETURNS:           None
CALLS:             Oracle, t_ins, t_cins, t_iins
CALLED BY:         main
*****************************************************************************/
{
   struct table t;
   struct t_node *tnode;
   struct t_col_info c;
   struct t_idx_info i;

   EXEC SQL DECLARE T_T1 CURSOR FOR                /* Get all the tables     */
      SELECT DISTINCT TABLE_NAME
      FROM   IAMS_TAB_INFO;

   EXEC SQL OPEN T_T1;
   for ( ; ; ) {
      EXEC SQL FETCH T_T1 INTO :tbl_name;
      if (sqlca.sqlcode == 1403)                   /* Any tables left?       */
         break;

      tbl_name.arr[tbl_name.len] = '\0';
      strcpy(t.name, tbl_name.arr);
      tnode = (struct t_node *) t_ins(&t_tree, t); /* Insert it in the tree  */

      EXEC SQL DECLARE T_C1 CURSOR FOR
         SELECT DISTINCT COLUMN_NAME, NULLABLE     /* Get the column info    */
         FROM   IAMS_TAB_INFO                      /* about the current      */
         WHERE  TABLE_NAME = :tbl_name;            /* table                  */
```

```
      EXEC SQL OPEN T_C1;
      for ( ; ; ) {
         EXEC SQL FETCH T_C1 INTO :col_name, :nullable;
         if (sqlca.sqlcode == 1403)                  /* Any columns left?       */
            break;

         EXEC SQL DECLARE T_C2 CURSOR FOR
            SELECT PRIMARY, FOREIGN, COMPOSITE       /* Get the key info about */
            FROM   IAMS_IND_INFO                     /* about the current       */
            WHERE  COLUMN_NAME = :col_name           /* column and table        */
            AND    TABLE_NAME  = :tbl_name;

         EXEC SQL OPEN T_C2;
         EXEC SQL FETCH T_C2 INTO :primary, :foreign, :composite;
         if (sqlca.sqlcode == 1403)                  /* Is it a key?            */
            strcpy(c.key, "---");                    /* NO:  Store blanks       */
         else {
            c.key[0] = primary;                      /* Primary?  Y/N           */
            c.key[1] = foreign;                      /* Foreign?  Y/N           */
            c.key[2] = composite;                    /* Composite?  Y/N         */
            c.key[3] = '\0';                         /* Terminate the string    */
         }
         EXEC SQL CLOSE T_C2;
         col_name.arr[col_name.len] = '\0';
         strcpy(c.name, col_name.arr);
         c.nullable = nullable;
         t_cins(&(tnode)->info, c);                  /* Insert the column into */
                                                     /* the current table node */
      }
      EXEC SQL CLOSE T_C1;

      EXEC SQL DECLARE T_I1 CURSOR FOR
         SELECT DISTINCT INDEX_NAME, UNIQUENESS      /* Get index info about   */
         FROM   IAMS_IND_INFO                        /* the current table      */
         WHERE  TABLE_NAME = :tbl_name;

      EXEC SQL OPEN T_I1;
      for ( ; ; ) {
         EXEC SQL FETCH T_I1 INTO :idx_name, :uniqueness;
         if (sqlca.sqlcode == 1403)                  /* Any indexes left?       */
            break;
         idx_name.arr[idx_name.len] = '\0';
         strcpy(i.name, idx_name.arr);
         i.unique = uniqueness;
         t_iins(&(tnode)->info, i);                  /* Insert the index into  */
                                                     /* the current table node */
      }
      EXEC SQL CLOSE T_I1;
   }
   EXEC SQL CLOSE T_T1;
}

get_column_info()
/*****************************************************************************
DESCRIPTION:      Loads column information about a database
```

```
                    into a tree strucuture ordered by
                   column_name.
PARAMETERS:        None
RETURNS:           None
CALLS:             Oracle, c_ins, c_tins, c_iins
CALLED BY:         main
*****************************************************************************/
{
   struct column c;
   struct c_tab_info t;
   struct c_idx_info i;
   struct c_node *cnode;

   EXEC SQL DECLARE C_C1 CURSOR FOR                    /* Get all columns      */
      SELECT DISTINCT COLUMN_NAME, DATA_TYPE, DATA_LENGTH, DATA_PRECISION
      FROM   IAMS_TAB_INFO;

   EXEC SQL OPEN C_C1;
   for ( ; ; ) {
      EXEC SQL FETCH C_C1 INTO :col_name, :type, :size, :decimal;
      if (sqlca.sqlcode == 1403)                       /* Any more columns?    */
         break;
      col_name.arr[col_name.len] = '\0';
      strcpy(c.name, col_name.arr);
      type.arr[type.len] = '\0';
      strcpy(c.type, type.arr);
      c.size = size;
      c.decimal = decimal;
      cnode = (struct c_node *) c_ins(&c_tree, c); /* Insert the column into */
                                                   /* the tree             */
      EXEC SQL DECLARE C_T1 CURSOR FOR
         SELECT DISTINCT TABLE_NAME, NULLABLE      /* Get all tables that  */
         FROM   IAMS_TAB_INFO                      /* contain the current  */
         WHERE COLUMN_NAME = :col_name;            /* column               */

      EXEC SQL OPEN C_T1;
      for ( ; ; ) {
         EXEC SQL FETCH C_T1 INTO :tbl_name, :nullable;
         if (sqlca.sqlcode == 1403)                /* Any more tables?     */
            break;
         tbl_name.arr[tbl_name.len] = '\0';
         strcpy(t.name, tbl_name.arr);
         t.nullable = nullable;

         EXEC SQL DECLARE C_T2 CURSOR FOR
            SELECT PRIMARY, FOREIGN, COMPOSITE     /* Get key info about the */
            FROM   IAMS_IND_INFO                   /* the current column and */
            WHERE  COLUMN_NAME = :col_name         /* table                */
            AND    TABLE_NAME  = :tbl_name;

         EXEC SQL OPEN C_T2;
         EXEC SQL FETCH C_T2 INTO :primary, :foreign, :composite;
         if (sqlca.sqlcode == 1403)                /* Is it a key?         */
            strcpy(t.key, "---");                  /* NO:  Store spaces    */
         else {
            t.key[0] = primary;                    /* Primary?  Y/N        */
```

```
            t.key[1] = foreign;                    /* Foreign?  Y/N          */
            t.key[2] = composite;                  /* Composite?  Y/N        */
            t.key[3] = '\0';                       /* Terminate the string   */
         }
         EXEC SQL CLOSE C_T2;
         c_tins(&cnode->info, t);                  /* Insert the table info  */
                                                   /* into the current column*/

         EXEC SQL DECLARE C_I1 CURSOR FOR
            SELECT DISTINCT INDEX_NAME             /* Get index info about   */
            FROM   IAMS_IND_INFO                   /* the current column and */
            WHERE  COLUMN_NAME = :col_name         /* table                  */
            AND    TABLE_NAME = :tbl_name;

         EXEC SQL OPEN C_I1;
         for ( ; ; ) {
            EXEC SQL FETCH C_I1 INTO :idx_name;
            if (sqlca.sqlcode == 1403)             /* Any more indexes?      */
               break;
            idx_name.arr[idx_name.len] = '\0';
            strcpy(i.name, idx_name.arr);
            c_iins(cnode->info.tbl_tail, i);       /* Insert the index info  */
                                                   /* into the current table */
         }
         EXEC SQL CLOSE C_I1;
      }
      EXEC SQL CLOSE C_T1;
   }
   EXEC SQL CLOSE C_C1;
}

get_index_info()
/*****************************************************************************
DESCRIPTION:      Loads index information about a database
                  into a tree structure ordered by
                  index_name.
PARAMETERS:       None
RETURNS:          None
CALLS:            Oracle, i_ins, i_cins
CALLED BY:        main
*****************************************************************************/
{
   struct index i;
   struct i_col_info c;
   struct i_node *inode;

   EXEC SQL DECLARE I_I1 CURSOR FOR
      SELECT DISTINCT INDEX_NAME, UNIQUENESS,      /* Get all indexes        */
             TABLE_NAME
      FROM   IAMS_IND_INFO;

   EXEC SQL OPEN I_I1;
   for ( ; ; ) {
      EXEC SQL FETCH I_I1 INTO :idx_name, :uniqueness, :tbl_name;
      if (sqlca.sqlcode == 1403)                   /* Last index found?      */
         break;
      idx_name.arr[idx_name.len] = '\0';
```

```
        strcpy(i.name, idx_name.arr);
        i.unique = uniqueness;
        tbl_name.arr[tbl_name.len] = '\0';
        strcpy(i.tbl_name, tbl_name.arr);
        inode = (struct i_node *) i_ins(&i_tree, i); /* Insert the index into  */
                                                     /* the tree               */
        EXEC SQL DECLARE I_C1 CURSOR FOR
           SELECT DISTINCT COLUMN_NAME               /* Get all column names   */
           FROM   IAMS_IND_INFO                      /* used to form the index */
           WHERE  INDEX_NAME = :idx_name;

        EXEC SQL OPEN I_C1;
        for ( ; ; ) {
           EXEC SQL FETCH I_C1 INTO :col_name;
           if (sqlca.sqlcode == 1403)                /* Any more columns?      */
              break;
           col_name.arr[col_name.len] = '\0';
           strcpy(c.name, col_name.arr);
           i_cins(&inode->info, c);                  /* Insert the column into */
                                                     /* the current index node */
        }
        EXEC SQL CLOSE I_C1;
     }
     EXEC SQL CLOSE I_I1;
}

save_user_info()
{
     fprintf(fd, "%s %s\n", userid.arr, password.arr);
}

save_table_info(t)
struct t_node *t;
/*****************************************************************************
DESCRIPTION:       Prints the table information stored in the
                   tree pointed to by t using an preorder
                   traversal.  The following information is
                   printed:  table name, columns used in the
                   table (along with key and nullable
                   information), and indexes are printed.
PARAMETERS:        struct t_node *t
RETURNS:           None
CALLS:             show_table_info, t_cnext, t_inext
CALLED BY:         main, show_table_info
*****************************************************************************/
{
     fprintf(fd,"%s\n",t->info.name);
     t->info.col_cur = t->info.col_head;           /* Reset column pointer   */
     while (t->info.col_cur != NULL) {
        fprintf(fd,"%s %s %c\n",
           t->info.col_cur->info.name,
           t->info.col_cur->info.key,
           t->info.col_cur->info.nullable);
        t_cnext(&(t)->info);                       /* Advance column pointer */
                                                   /* to the next node       */
     }
```

```
    fprintf(fd,"\n");
   t->info.idx_cur = t->info.idx_head;              /* Reset index pointer    */
   while (t->info.idx_cur != NULL) {
      fprintf(fd,"%s %c\n",
         t->info.idx_cur->info.name,
         t->info.idx_cur->info.unique);
      t_inext(&(t)->info);                          /* Advance index pointer  */
                                                    /* to the next node       */
   }
   fprintf(fd,"\n");
   if (t->lchild != NULL)                           /* If NOT NULL, call for  */
      save_table_info(t->lchild);                   /* the left sub-tree      */
   if (t->rchild != NULL)                           /* If NOT NULL, call for  */
      save_table_info(t->rchild);                   /* the right sub-tree     */
}

save_column_info(c)
struct c_node *c;
/*****************************************************************************
DESCRIPTION:       Prints the column information stored in the
                   tree pointed to by c using an preorder
                   traversal.  The following information is
                   printed:  column name, type, size, number
                   of decimals, all tables the column is used
                   in, and all indexes the column is used in.
PARAMETERS:        struct c_node *c
RETURNS:           None
CALLS:             show_column_info, c_inext, c_tnext
CALLED BY:         main, show_column_info
*****************************************************************************/
{
   fprintf(fd,"%s %s %d %d\n",
      c->info.name, c->info.type,
      c->info.size, c->info.decimal);
   c->info.tbl_cur = c->info.tbl_head;              /* Reset table pointer    */
   while (c->info.tbl_cur != NULL) {
      fprintf(fd,"%s %s %c\n",
         c->info.tbl_cur->info.name,
         c->info.tbl_cur->info.key,
         c->info.tbl_cur->info.nullable);
                                                    /* Reset index pointer    */
      c->info.tbl_cur->info.idx_cur = c->info.tbl_cur->info.idx_head;
      while (c->info.tbl_cur->info.idx_cur != NULL) {
         fprintf(fd,"%s\n",
            c->info.tbl_cur->info.idx_cur->info.name);
         c_inext(&c->info.tbl_cur->info);           /* Advance index pointer  */
                                                    /* to the next node       */
      }
      c_tnext(&c->info);                            /* Advance table pointer  */
                                                    /* to the next node       */
      fprintf(fd, "\n");
   }
   fprintf(fd,"\n");
   if (c->lchild != NULL)                           /* If NOT NULL, call for  */
      save_column_info(c->lchild);                  /* the left sub-tree      */
   if (c->rchild != NULL)                           /* If NOT NULL, call for  */
```

```
        save_column_info(c->rchild);                    /* the right sub-tree     */
}

save_index_info(i)
struct i_node *i;
/*****************************************************************************
DESCRIPTION:        Prints the index information stored in the
                    tree pointed to by i using an inorder
                    traversal.  The following information is
                    printed:  index name, uniqueness, table
                    name where the index is used, and all
                    columns used in the index.
PARAMETERS:         struct i_node *i
RETURNS:            None
CALLS:              show_index_info, i_cnext
CALLED BY:          main, show_index_info
*****************************************************************************/
{
   fprintf(fd,"%s %s %c\n",
      i->info.name, i->info.tbl_name, i->info.unique);
   i->info.col_cur = i->info.col_head;              /*Reset the column pointer*/
   while (i->info.col_cur != NULL) {
      fprintf(fd,"%s\n",i->info.col_cur->info.name);
      i_cnext(&i->info);                            /* Advance the column ptr */
                                                    /* to the next node       */
   }
   fprintf(fd,"\n");
   if (i->lchild != NULL)                           /* If NOT NULL, call for  */
      save_index_info(i->lchild);                   /* the left sub-tree      */
   if (i->rchild != NULL)                           /* If NOT NULL, call for  */
      save_index_info(i->rchild);                   /* the right sub-tree     */
}

ksquit()
{
   EXEC SQL COMMIT WORK RELEASE;
   if (sqlca.sqlcode != 0)                          /* Error committing?      */
      printf("(KS) Unable to commit work.\n");
   del_trees();                                     /* Delete all the trees   */
}

ksinit()
/*****************************************************************************
DESCRIPTION:        Connects to Oracle.  Loads information
                    needed for the knowledge source from the
                    IAMS tables in Oracle.  Displays the
                    information.  Deletes the trees containing
                    the information, and disconnects from
                    Oracle.
PARAMETERS:         None
RETURNS:            None
CALLS:              Oracle, oracle_login, ini_trees,
                    get_table_info, get_column_info,
                    get_index_info, show_table_info,
                    show_column_info, show_index_info,
                    del_trees
```

```
CALLED BY:        None
*****************************************************************************/
{
   prompt();
   if (oracle_login() != 0) {                          /* Error connecting?      */
      printf("(KS) Unable to connect to Oracle with %s/%s\n",
         userid.arr,password.arr);
      exit(0);
   }
   init_trees();                                       /* Initialize the trees   */
   get_table_info();                                   /* Load table information */
   get_column_info();                                  /* Load column info       */
   get_index_info();                                   /* Load index information */
   save_user_info();                                   /* Save UID and PWD       */
   save_table_info(t_tree);                            /* Save the table info    */
   fprintf(fd,"\n");
   save_column_info(c_tree);                           /*  and the column info   */
   fprintf(fd,"\n");
   save_index_info(i_tree);                            /*  and the index info to */
   fprintf(fd,"\n");
                                                       /*  the KS data file      */
}

load_table_info()
{
   char buf[80];
   struct table t;
   struct t_node *tnode;
   struct t_col_info c;
   struct t_idx_info i;

   for ( ; ; ) {
      fgets(t.name, 31, fd);
      t.name[strlen(t.name)-1] = '\0';
      if (t.name[0] == '\0')
         break;
      tnode = (struct t_node *) t_ins(&t_tree, t);
      for ( ; ; ) {
         fgets(buf, 80, fd);
         buf[strlen(buf)-1] = '\0';
         if (buf[0] == '\0')
            break;
         c.nullable = buf[strlen(buf)-1];
         strcpy(c.name, strtok(buf, " "));
         strcpy(c.key, strtok(NULL, " "));
         t_cins(&tnode->info, c);
      }
      for ( ; ; ) {
         fgets(buf, 80, fd);
         buf[strlen(buf)-1] = '\0';
         if (buf[0] == '\0')
            break;
         i.unique = buf[strlen(buf)-1];
         strcpy(i.name, strtok(buf, " "));
         t_iins(&tnode->info, i);
      }
```

```
    }
}

load_column_info()
{
   struct column c;
   struct c_tab_info t;
   struct c_idx_info i;
   struct c_node *cnode;
   char buf[80];

   for ( ; ; ) {
      fgets(buf, 80, fd);
      buf[strlen(buf)-1] = '\0';
      if (buf[0] == '\0')
         break;
      strcpy(c.name, strtok(buf, " "));
      strcpy(c.type, strtok(NULL, " "));
      c.size = atoi(strtok(NULL, " "));
      c.decimal = atoi(strtok(NULL, " "));
      cnode = (struct c_node *) c_ins(&c_tree, c);
      for ( ; ; ) {
         fgets(buf, 80, fd);
         buf[strlen(buf)-1] = '\0';
         if (buf[0] == '\0')
            break;
         t.nullable = buf[strlen(buf)-1];
         strcpy(t.name, strtok(buf, " "));
         strcpy(t.key, strtok(NULL, " "));
         c_tins(&cnode->info, t);
         for ( ; ; ) {
            fgets(i.name, 31, fd);
            i.name[strlen(i.name)-1] = '\0';
            if (i.name[0] == '\0')
               break;
            c_iins(cnode->info.tbl_tail, i);
         }
      }
   }
}

load_index_info()
{
   struct index i;
   struct i_col_info c;
   struct i_node *inode;
   char buf[80];

   for ( ; ; ) {
      fgets(buf, 80, fd);
      buf[strlen(buf)-1] = '\0';
      if (buf[0] == '\0')
         break;
      i.unique = buf[strlen(buf)-1];
      strcpy(i.name, strtok(buf, " "));
      strcpy(i.tbl_name, strtok(NULL, " "));
```

```
        inode = (struct i_node *) i_ins(&i_tree, i);
        for ( ; ; ) {
           fgets(c.name, 31, fd);
           c.name[strlen(c.name)-1] = '\0';
           if (c.name[0] == '\0')
              break;
           i_cins(&inode->info, c);
        }
     }
}

load_user_info()
{
   char buf[80];

   fgets(buf, 80, fd);
   buf[strlen(buf)-1] = '\0';
   strcpy(userid.arr, strtok(buf, " "));
   strcpy(password.arr, strtok(NULL, " "));
   userid.len = strlen(userid.arr);
   password.len = strlen(password.arr);
}

ksload()
/*****************************************************************************
DESCRIPTION:
PARAMETERS:       None
RETURNS:          None
CALLS:            None
CALLED BY:        None
*****************************************************************************/
{
   load_user_info();                                  /* Load user id info      */
   if (oracle_login() != 0) {                         /* Error connecting?      */
      printf("(KS) Unable to connect to Oracle with %s/%s\n",
         userid.arr,password.arr);
      exit(0);
   }
   init_trees();                                      /* Initialize the trees   */
   load_table_info();                                 /* Load table information */
   load_column_info();                                /* Load column info       */
   load_index_info();                                 /* Load index information */
}

main(argc, argv)
int argc;
char *argv[];
{
   char t1[30], t1op[5], t1range[30], t1table[30], t2[30], t2op[5], t2range[30],
      t2table[30],jc[30],jcop[30],jcrange[30],qry[4096],fname[50], resname[50];
   int res = NO_DATA, procres = 0;

   iams_log = fopen("iams_log.txt", "at");
   if (argc > 1)
      strcpy(app_name, argv[1]);
   else {
```

```
        fprintf(iams_log,"KS:  Application name is not present.\n");
        fclose(iams_log);
        exit(-1);
    }

    iams_qry = fopen("iams_qry.txt", "at");

    strlwr(app_name);
    sprintf(resname,"iams_%s_results.txt",app_name);
    iams_res = fopen(resname,"rt");
    if (fgets(t1, 31, iams_res) != NULL)
       t1[strlen(t1)-1] = '\0';
    else {
       fprintf(iams_log,"KS:  Error reading %s.\n",resname);
       fclose(iams_log);
       exit(-1);
    }
    if (fgets(tlop, 31, iams_res) != NULL)
       tlop[strlen(tlop)-1] = '\0';
    else {
       fprintf(iams_log,"KS:  Error reading %s.\n",resname);
       fclose(iams_log);
       exit(-1);
    }
    if (fgets(tlrange, 31, iams_res) != NULL)
       tlrange[strlen(tlrange)-1] = '\0';
    else {
       fprintf(iams_log,"KS:  Error reading %s.\n",resname);
       fclose(iams_log);
       exit(-1);
    }
    if (fgets(t2, 31, iams_res) != NULL)
       t2[strlen(t2)-1] = '\0';
    else {
       fprintf(iams_log,"KS:  Error reading %s.\n",resname);
       fclose(iams_log);
       exit(-1);
    }
    if (fgets(t2op, 31, iams_res) != NULL)
       t2op[strlen(t2op)-1] = '\0';
    else {
       fprintf(iams_log,"KS:  Error reading %s.\n",resname);
       fclose(iams_log);
       exit(-1);
    }
    if (fgets(t2range, 31, iams_res) != NULL)
       t2range[strlen(t2range)-1] = '\0';
    else {
       fprintf(iams_log,"KS:  Error reading %s.\n",resname);
       fclose(iams_log);
       exit(-1);
    }
    fclose(iams_res);

    strupr(t1);
    strupr(t1cp);
```

```
    strupr(t2);
   strupr(t2op);
   t1table[0] = '\0';
   t2table[0] = '\0';
   jc[0] = '\0';
   jcop[0] = '\0';
   jcrange[0] = '\0';

#ifdef Debug
   printf("Target 1:  %s { %s %s }\n",t1,t1op,t1range);
   printf("Target 2:  %s { %s %s }\n",t2,t2op,t2range);
#endif

   iams_res = fopen(resname,"wt");
   sprintf(fname,"iams_%s_ks.txt", app_name);
   fd = fopen(fname, "rt");
   if (fd == NULL) {
      fd = fopen(fname, "wt");
      ksinit();
   }
   else ksload();

   res = find_relation(t1, t2, t1table, jc, t2table);
   switch (res) {
      case NO_DATA:
         fprintf(iams_log, "%s:  %s and %s are not related.\n",
            app_name, t1, t2);
         fprintf(iams_res, "%d\n",NOTREL);
         break;
      case NOT_FOUND:
         fprintf(iams_log, "%s:  Either %s or %s does not exist in the KS.\n",
            app_name, t1, t2);
         fprintf(iams_log, "This indicates an inconsistency in the KS/MKS.\n");
         break;
      case TBL:
         if ((t1[0] != NULL) && (t2[0] != NULL))
            fprintf(iams_log, "%s:  Both %s and %s reside in %s.\n",
               app_name, t1, t2, t1table);
         else if (t1[0] != NULL)
            fprintf(iams_log, "%s:  %s resides in %s.\n",
               app_name, t1, t1table);
         else
            fprintf(iams_log, "%s:  %s resides in %s.\n",
               app_name, t2, t1table);
         break;
      case TBL_COL_TBL:
         fprintf(iams_log, "%s:  A JOIN is needed.\n", app_name);
         fprintf(iams_log, "\t%s is in %s, and\n",t1, t1table);
         fprintf(iams_log, "\t%s is in %s, and\n",t2, t2table);
         fprintf(iams_log, "\tthe JOIN column is %s.\n",jc);
         break;
      default:
         fprintf(iams_log, "Unknown results returned by simple_case2(..).\n");
         fprintf(iams_log, "Return value was %d.\n",res);
   }
   if ((res != NO_DATA) && (res != NOT_FOUND)) {
```

```
    bld_query(qry, res, t1, t1op, t1range, t1table, t2, t2op, t2range,
      t2table, jc, jcop, jcrange);                          /* build query*/
   fprintf(iams_res, "%d\n",OK);
   if ((procres = process_sql(qry)) == -1) {
      fprintf(iams_log, "%s:  Error processing the following query:\n",
         app_name);
      fprintf(iams_log, "%s\n\n",qry);
   }
   else if (procres == 0) {
      fprintf(iams_log,"KS:  Query returned no results.\n");
      unlink(resname);
      iams_res = fopen(resname,"wt");
      fprintf(iams_res, "%d\n",OKNOROWS);
   }
   } /* End If */
   ksquit();
   fclose(fd);
   fclose(iams_res);
   fclose(iams_qry);
   fclose(iams_log);
}
```

Appendix C Pro*C Code

Oct 11 17:40 1992 mks.pc Page 1

```
/*****************************************************************************
                                PROGRAM MKS.PC
   Pro*C code developed to construct and manipulate IAMS meta knowledge source.
   Project specifications developed by:
         Peggy Wright in support of IAMS research
   Code written by:
         Shannon Thornton              18 AUG 92

                              Naming Convention
   Functions used for manipulating the tree or list structure begin with the
   first letter of the structure used.  For example, to insert a column into
   the column tree, c_ins(..) is called.  The c_ indicates the function uses
   the column tree structure.  All functions that manipulate the table list
   inside a node in the column tree begin with c_t.  Any function that
   manipulates the index list inside the previously mentioned table list
   begins with c_i.  All table tree functions start with t_, and all index
   tree functions start with i_.  This convention is also used for global
   tree variables, c_tree, t_tree, and i_tree, which reference the column
   tree, table tree, and index tree, respectively.
*****************************************************************************/

#include <stdio.h>
#include <string.h>
#include "mks.h"

#define USERID "iams"
#define PASSWORD "iammks"

extern struct m_node *m_tree;       /* Pointer to the index tree              */

FILE *fd;                           /* File pointer for KS data file          */
extern FILE *iams_log;              /* File pointer to the IAMS log file      */

EXEC SQL BEGIN DECLARE SECTION;
   VARCHAR userid[20];              /* User id used to connect to Oracle      */
   VARCHAR password[20];            /* Password used to connect to Oracle     */
/* - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - */
/* The following host variables are used for storing data returned from      */
/* embedded SQL Queries.                                                    */
/* - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - */
   VARCHAR col_name[31];
   VARCHAR app_name[31];
   int decimal;
   int size;
   VARCHAR type[10];
EXEC SQL END DECLARE SECTION;
EXEC SQL INCLUDE SQLCA;

int oracle_login()
/*****************************************************************************
DESCRIPTION:      Prompts for a userid and password.  Once
                  this is entered, it tries to connect to
                  Oracle using the userid and password.
PARAMETERS:       None
RETURNS:          Oracle error code (0 if no error)
CALLS:            Oracle
```

```
CALLED BY:        main
*****************************************************************************/
{
   strcpy(userid.arr, USERID);
   strcpy(password.arr, PASSWORD);
   userid.len = strlen(userid.arr);
   password.len = strlen(password.arr);
   EXEC SQL CONNECT :userid          /* Connect to Oracle using the entered    */
   IDENTIFIED BY :password;          /* user ID and password                   */
   return sqlca.sqlcode;             /* Return the Oracle error code           */
}

init_tree()
/*****************************************************************************
DESCRIPTION:      Initializes the table, column, and index
                  trees.
PARAMETERS:       None
RETURNS:          None
CALLS:            t_init, c_init, i_init
CALLED BY:        main
*****************************************************************************/
{
   m_init();                         /* Initialize the meta-KS tree, m_tree    */
}

del_tree()
/*****************************************************************************
DESCRIPTION:      Deletes the table, column, and index
                  trees.
PARAMETERS:       None
RETURNS:          None
CALLS:            t_deltree, c_deltree, i_deltree
CALLED BY:        main
*****************************************************************************/
{
   m_deltree(m_tree);                /* Delete the MKS tree                    */
}

get_mks_info()
/*****************************************************************************
DESCRIPTION:
PARAMETERS:       None
RETURNS:          None
CALLS:            Oracle, c_ins, c_tins, c_iins
CALLED BY:        main
*****************************************************************************/
{
   struct mks m;
   struct m_app_info a;
   struct m_node *mnode;

   EXEC SQL DECLARE M_M1 CURSOR FOR              /* Get all columns         */
      SELECT DISTINCT COLUMN_NAME, DATA_TYPE, DATA_LENGTH, DATA_PRECISION
      FROM   MKS_TAB_INFO;

   EXEC SQL OPEN M_M1;
```

```
    for ( ; ; ) {
      EXEC SQL FETCH M_M1 INTO :col_name, :type, :size, :decimal;
      if (sqlca.sqlcode == 1403)                    /* Any more columns?      */
         break;
      col_name.arr[col_name.len] = '\0';
      strcpy(m.name, col_name.arr);
      type.arr[type.len] = '\0';
      strcpy(m.type, type.arr);
      m.size = size;
      m.decimal = decimal;
#ifdef Debug
      printf("----------- %s %s\n", m.name, m.type);
#endif
      mnode = (struct m_node *) m_ins(&m_tree, m); /* Insert the column into */
                                                   /* the tree               */
      EXEC SQL DECLARE M_A1 CURSOR FOR
         SELECT DISTINCT APPLICATION               /* Get all tables that    */
         FROM   MKS_TAB_INFO                       /* contain the current    */
         WHERE COLUMN_NAME = :col_name;            /* column                 */

      EXEC SQL OPEN M_A1;
      for ( ; ; ) {
         EXEC SQL FETCH M_A1 INTO :app_name;
         if (sqlca.sqlcode == 1403)                /* Any more tables?       */
            break;
         app_name.arr[app_name.len] = '\0';
         strcpy(a.name, app_name.arr);
#ifdef Debug
      printf("------------- %s\n", a.name);
#endif
         m_ains(&mnode->info, a);                  /* Insert the table info  */
                                                   /* into the current column*/
      }
      EXEC SQL CLOSE M_A1;
   }
   EXEC SQL CLOSE M_M1;
}

load_mks_info()
/*****************************************************************************
DESCRIPTION:
PARAMETERS:       None
RETURNS:          None
CALLS:            Oracle, c_ins, c_tins, c_iins
CALLED BY:        main
*****************************************************************************/
{
   struct mks m;
   struct m_app_info a;
   struct m_node *mnode;
   char buf[80];

   for ( ; ; ) {
      if (fgets(buf, 80, fd) == NULL)
         break;
      buf[strlen(buf)-1] = '\0';
```

```
        strcpy(m.name, strtok(buf, " "));
        strcpy(m.type, strtok(NULL, " "));
        m.size = atoi(strtok(NULL, " "));
        m.decimal = atoi(strtok(NULL, " "));
        mnode = (struct m_node *) m_ins(&m_tree, m); /* Insert the column into */
                                                     /* the tree               */
        for ( ; ; ) {
           if (fgets(a.name, 31, fd) == NULL)
              break;
           a.name[strlen(a.name)-1] = '\0';
           if (a.name[0] == '\0')
              break;
           m_ains(&mnode->info, a);                  /* Insert the table info  */
                                                     /* into the current column*/
        }
     }
}

save_mks_info(m)
struct m_node *m;
{
   fprintf(fd, "%s %s %d %d\n", m->info.name, m->info.type,
      m->info.size, m->info.decimal);
   m->info.app_cur = m->info.app_head;
   while (m->info.app_cur != NULL) {
      fprintf(fd, "%s\n", m->info.app_cur->info.name);
      m_anext(&(m)->info);
   }
   fprintf(fd, "\n");
   if (m->lchild != NULL)
      save_mks_info(m->lchild);
   if (m->rchild != NULL)
      save_mks_info(m->rchild);
}

mksinit()
/*****************************************************************************
DESCRIPTION:      Connects to Oracle.  Loads information
                  needed for the knowledge source from the
                  IAMS tables in Oracle.  Displays the
                  information.  Deletes the trees containing
                  the information, and disconnects from
                  Oracle.
PARAMETERS:       None
RETURNS:          None
CALLS:            Oracle, oracle_login, ini_trees,
                  get_table_info, get_column_info,
                  get_index_info, show_table_info,
                  show_column_info, show_index_info,
                  del_trees
CALLED BY:        None
*****************************************************************************/
{
#ifdef DEBUG
   printf("-------- Trying to connect to Oracle\n");
#endif
```

```
    if (oracle_login() != 0) {                          /* Error connecting?      */
      fprintf(iams_log, "(MKS) Unable to connect to Oracle with %s/%s\n",
         userid.arr,password.arr);
      exit(0);
   }
   init_tree();                                         /* Initialize the trees   */
#ifdef DEBUG
   printf("-------- Loading MKS from Oracle\n");
#endif
   get_mks_info();                                      /* Load MKS info          */
#ifdef DEBUG
   printf("-------- Saving MKS iams_mks.txt\n");
#endif
   save_mks_info(m_tree);
#ifdef DEBUG
   printf("-------- Exitting from Oracle\n");
#endif
   EXEC SQL COMMIT WORK RELEASE;
   if (sqlca.sqlcode != 0) {
      fprintf(iams_log, "(MKS) Unable to commit work.\n");
      fprintf(stderr, "(MKS) Unable to commit work.\n");
   }
}

mksload()
{
   init_tree();
   load_mks_info();
}

mksstart()
{
#ifdef DEBUG
   printf("-------- Checking for iams_mks.txt\n");
#endif
   if ((fd = fopen("iams_mks.txt", "rt")) == NULL) {
#ifdef DEBUG
      printf("-------- Did not find iams_mks.txt\n");
#endif
      fd = fopen("iams_mks.txt", "wt");
      mksinit();
   }
   else mksload();
}

mksquit()
{
   del_tree();
   fclose(fd);
}
```

APPENDIX D

C CODE

Appendix D C Code

IAMS C Source Code

The purpose of this table is to describe the functionality of the individual C code modules contained in the IAMS system.

FILE	DESCRIPTION
ATTACH.C	**Procedural Attachments:** This file contains the intelligence of the Knowledge Source. ATTACH.C locates the target column(s) in the KS and tries to establish a link if more than one target column is given. If both target columns are in the same table, no link is needed; however, if the target columns are in different tables, a join column is found, if one exists. The results are returned to KS.PC to be passed on to QC.C.
C_TREE.C	**Column Tree:** Contains tree and list routines to implement the column structure. This structure consists of a binary tree where each node in the tree contains column information and a linked list of tables in which the column occurs. Each node in the table list contains table information and a linked list of index names that occur in the corresponding table.
I_TREE.C	**Index Tree:** Contains tree and list routines to implement the index structure. This structure consists of a binary tree where each node in the tree contains index information, a table name, and a linked list of column names comprising the corresponding index.
IAMS_CONST.H	**IAMS Constants:** This is a header file containing system definitions and constants used in both the KS and MKS.
IUI.C	**Intelligent User Interface:** This file contains the function main. It contains all code for creating the user interface and the intelligence of the MKS. Once the user enters the target column(s) and search constraints, he/she presses the EXECUTE button. The MKS is searched to find the location of the target column(s). If there is more than one target column, the MKS determines if they are in the same KS. If so, a single KS is forked. If the target columns are in different KS's, the user is prompted to choose a potential join column. Then, two KS's are forked. The query and query results are displayed.
M_TREE.C	**MKS Tree:** This file contains the code for creating and manipulating the MKS tree. Each node contains a column name and a list of application names or KS names. This is needed to determine which columns can be used to link two KSs.
MKS.H	**MKS Data Types:** Contains the data type definitions used by the MKS.
QC.C	**Query Constructor:** This code is called from KS.PC. The target column(s) and search constraints are provided along with any necessary join columns. The Query Constructor builds an SQL query a returns it in a character string.
STRUTIL.C	**String Utilities:** Contains several helpful string utilities. These include function for performing case conversion and removing leading and/or trailing spaces.

Appendix D C Code

FILE	DESCRIPTION
T_TREE.C	**Table Tree:** Contains tree and list routines to implement the table structure. This structure consists of a binary tree where each node in the tree contains table information, a linked list of column names which occur in the table, and a linked list of index names which occur in the table. Each node in the column list contains column information, while each node in the index list contains index information.
TYPES.H	**Data Types:** This is an include file containing the data structure definitions for the KS.

Appendix D C Code

Oct 11 17:46 1992 attach.c Page 1

```
/*****************************************************************************
                                PROGRAM ATTACH.C
   Code used to determine column locations, target relationships, and search
         priorities by searching through KS tree structures.
   Specification designed by:
         Peggy Wright in support of IAMS research
   Coded by:
         Shannon Thornton                    15 AUG 92
*****************************************************************************/
#include <stdio.h>
#include <string.h>
#include "types.h"

extern struct c_node *c_tree;
extern struct t_node *t_tree;
extern struct i_node *i_tree;

extern FILE *iams_log;
extern FILE *iams_res;
extern char app_name[NAMELENGTH];

int priority(key, nullable, idx_head)
char key[4], nullable;
struct c_inode *idx_head;
{
   if ((key[0] == 'Y') && (key[1] == 'N'))
      return LEVEL6;   /* Primary -- Not foreign */
   else if ((key[0] == 'Y') && (key[1] == 'Y'))
      return LEVEL5;   /* Primary and Foreign */
   else if (key[1] == 'Y')
      return LEVEL4;   /* Foreign -- Not primary */
   else if (key[0] == 'N')
      return LEVEL3;   /* It is a key -- Just Not primary or foriegn */
   else if (idx_head != NULL)
      return LEVEL2;   /* It's in an index -- Not primary or foreign */
   else if (nullable == 'N')
      return LEVEL1;   /* Not Nullable */
   else return LEVEL0; /* It exists */
}

find_relation(t1, t2, t1table, jc, t2table)
char *t1, *t2, *t1table, *jc, *t2table;
{
   int tpri = NOPRI, tpri2 = NOPRI, pri = NOPRI, tpri3 = NOPRI;
   struct c_node *t1cnode, *t2cnode;
   struct i_node *t1inode, *t2inode;
   struct t_node *t1tnode, *t2tnode;
   struct c_tab_info *t1tabinfo, *t2tabinfo;
   struct t_col_info t1colinfo, t2colinfo;

/*------------- Be sure both columns exist before proceeding. ---------------*/
   t1cnode = (struct c_node *) c_find(c_tree, t1);
   t2cnode = (struct c_node *) c_find(c_tree, t2);
   if ((t1[0] != NULL) && (t1cnode == NULL)) {
      fprintf(iams_log,"%s:  Could not find %s in the KS.\n",
         app_name, t1);
```

```
        return NOT_FOUND;
    }
    if ((t2[0] != NULL) && (t2cnode == NULL)) {
       fprintf(iams_log,"%s:  Could not find %s in the KS.\n",
          app_name, t2);
       return NOT_FOUND;
    }

    tltable[0] = '\0';
/*----------------- See if only one column was passed in. -------------------*/
    if ((tlcnode == NULL) && (t2cnode != NULL)) {
       t2cnode->info.tbl_cur = t2cnode->info.tbl_head;
       while (t2cnode->info.tbl_cur != NULL) {
          if ((tpri = priority(t2cnode->info.tbl_cur->info.key,
                t2cnode->info.tbl_cur->info.nullable,
                t2cnode->info.tbl_cur->info.idx_head)) > pri) {
             pri = tpri;
             strcpy(tltable,t2cnode->info.tbl_cur->info.name);
             printf("KS:  %s is a priority %d table for %s.\n",tltable,pri,t1);
          }
          c_tnext(&t2cnode->info);
       }
       return TBL;
    }

    if ((t2cnode == NULL) && (tlcnode != NULL)) {
       tlcnode->info.tbl_cur = tlcnode->info.tbl_head;
       while (tlcnode->info.tbl_cur != NULL) {
          if ((tpri = priority(tlcnode->info.tbl_cur->info.key,
                tlcnode->info.tbl_cur->info.nullable,
                tlcnode->info.tbl_cur->info.idx_head)) > pri) {
             pri = tpri;
             strcpy(tltable,tlcnode->info.tbl_cur->info.name);
             printf("KS:  %s is a priority %d table for %s.\n",tltable,pri,t1);
          }
          c_tnext(&tlcnode->info);
       }
       return TBL;
    }

/*-- Try to find a match in the table lists.  If found return best match. --*/
    tlcnode->info.tbl_cur = tlcnode->info.tbl_head;
    while (tlcnode->info.tbl_cur != NULL) {
       t2cnode->info.tbl_cur = t2cnode->info.tbl_head;
       while (t2cnode->info.tbl_cur != NULL) {
          if (strcmp(tlcnode->info.tbl_cur->info.name,
              t2cnode->info.tbl_cur->info.name) == 0)
             if (((tpri = priority(tlcnode->info.tbl_cur->info.key,
                   tlcnode->info.tbl_cur->info.nullable,
                   tlcnode->info.tbl_cur->info.idx_head)) > pri) ||
                 ((tpri2 = priority(t2cnode->info.tbl_cur->info.key,
                   t2cnode->info.tbl_cur->info.nullable,
                   t2cnode->info.tbl_cur->info.idx_head)) > pri)) {
                pri = max(tpri,tpri2);
                strcpy(tltable,tlcnode->info.tbl_cur->info.name);
                printf("KS:  %s is a priority %d table for %s and %s.\n",
```

```
                    t1table,pri,t1,t2);
            }
         c_tnext(&t2cnode->info);
      }
      c_tnext(&t1cnode->info);
   }
   if (t1table[0] != '\0')
      return TBL;

/*--------------------- Find best potential join columns ----------------------*/
   t1cnode->info.tbl_cur = t1cnode->info.tbl_head;
   while (t1cnode->info.tbl_cur != NULL) {
      t2cnode->info.tbl_cur = t2cnode->info.tbl_head;
      while (t2cnode->info.tbl_cur != NULL) {
         if ((t1tnode = (struct t_node *) t_find(t_tree,
              t1cnode->info.tbl_cur->info.name)) == NULL) {
            fprintf(iams_log,"%s:  Inconsistency in the KS.\n", app_name);
            fprintf(iams_log,"%s is listed as a table for %s,\n",
               t1cnode->info.tbl_cur->info.name, t1cnode->info.name);
            fprintf(iams_log,"but this table was not found in the table tree.\n");
         }
         else if ((t2tnode = (struct t_node *) t_find(t_tree,
                   t2cnode->info.tbl_cur->info.name)) == NULL) {
            fprintf(iams_log,"%s:  Inconsistency in the KS.\n", app_name);
            fprintf(iams_log,"%s is listed as a table for %s,\n",
               t2cnode->info.tbl_cur->info.name, t2cnode->info.name);
            fprintf(iams_log,"but this table was not found in the table tree.\n");
         }
         else {
            t1tnode->info.col_cur = t1tnode->info.col_head;
            while (t1tnode->info.col_cur != NULL) {
               t2tnode->info.col_cur = t2tnode->info.col_head;
               while (t2tnode->info.col_cur != NULL) {
                  if (strcmp(t1tnode->info.col_cur->info.name,
                      t2tnode->info.col_cur->info.name) == 0) {
                     t1colinfo = t1tnode->info.col_cur->info;
                     t2colinfo = t2tnode->info.col_cur->info;
               /* Find JC priority */
                        tpri = priority(t1colinfo.key,t1colinfo.nullable,
                           t1tnode->info.idx_head);
                        tpri2 = priority(t2colinfo.key,t2colinfo.nullable,
                           t2tnode->info.idx_head);
                        tpri3 = max(tpri,tpri2) * 10;
               /* Find the column(s) priority */
                        tpri = priority(t1cnode->info.tbl_cur->info.key,
                           t1cnode->info.tbl_cur->info.nullable,
                           t1cnode->info.tbl_cur->info.idx_head);
                        tpri2 = priority(t2cnode->info.tbl_cur->info.key,
                           t2cnode->info.tbl_cur->info.nullable,
                           t2cnode->info.tbl_cur->info.idx_head);
                        tpri3 = tpri3 + max(tpri,tpri2);
                        if (tpri3 > pri) {
                           pri = tpri3;
                           strcpy(t1table,t1tnode->info.name);
                           strcpy(jc,t1colinfo.name);
                           strcpy(t2table,t2tnode->info.name);
```

```
                        printf("KS:  %s is a priority %d jc for %s and %s.\n",
                          jc,pri,t1,t2);
                    }
                }
                t_cnext(&t2tnode->info);
            }
            t_cnext(&t1tnode->info);
        }
      }
      c_tnext(&t2cnode->info);
    }
    c_tnext(&t1cnode->info);
  }
  if (t1table[0] != NULL)
    return TBL_COL_TBL;
  else return NO_DATA;
}
```

Appendix D C Code

Oct 11 17:46 1992 c_tree.c Page 1

```
/*****************************************************************************
                                    PROGRAM C_TREE.C
   Project specifications developed by:
         Peggy Wright in support of IAMS research
   Code written by:
         Shannon Thornton           03 AUG 92

   Contains the necessary routines to create, manipulate, and dispose of a
   binary tree.  Each node in the tree contains a column name, data type,
   data size, number of decimal places (if numeric), and a linked list of
   tables.  The list contains all table names in which the column is used.
   There is a linked list stored in each node of the table list.  This list
   contains the names of all indexes used in the current table that reference
   the current column.  See TYPES.H for the structure definitions.

                                Naming Convention
   Functions used for manipulating the tree or list structure begin with the
   first letter of the structure used.  For example, to insert a column into
   the column tree, c_ins(..) is called.  The c_ indicates the function uses
   the column tree structure.  All functions that manipulate the table list
   inside a node in the column tree begin with c_t.  Any function that
   manipulates the index list inside the previously mentioned table list
   begins with c_i.  All table tree functions start with t_, and all index
   tree functions start with i_.  This convention is also used for global
   tree variables, c_tree, t_tree, and i_tree, which reference the column
   tree, table tree, and index tree, respectively.
*****************************************************************************/

#include <string.h>
#include <stdio.h>
#include <malloc.h>
#include "types.h"

/* - - - - - - - - - - - - - * Global Tree Variables * - - - - - - - - - - - - - */
struct c_node *c_tree;      /* Pointer to the column tree                      */
int cmp;                    /* Temporary variable used in find and insert      */
                            /* Since find and insert are recursive procedures */
                            /* the temporary variable is declared global.  If */
                            /* it was declared locally, it would be            */
                            /* re-allocated on the stack for every function    */
                            /* call.                                           */


/* - - - - - - - - - - - - - * Function definitions * - - - - - - - - - - - -  */

/* - - - - - - - - - - - - - - - * * * * * * * * * * - - - - - - - - - - - - - - - */
/* - - - - - - - - - - - - - - - * Tree  Functions * - - - - - - - - - - - - - - - */
/* - - - - - - - - - - - - - - - * * * * * * * * * * - - - - - - - - - - - - - - - */
c_shownode(info)
struct column info;
/*****************************************************************************
DESCRIPTION:      Prints the contents of info to standard
                  output.
PARAMETERS:       struct column info
RETURNS:          None
CALLS:            None
```

```
CALLED BY:          c_lnr
*****************************************************************************/
{
   printf("%-30s %-9s %3d ",info.name, info.type, info.size);
   if (info.type[0] == 'N') printf("%d\n",info.decimal);
   else printf("\n");
}

c_init()
/*****************************************************************************
DESCRIPTION:        Initializes the global tree variable c_tree
                    to NULL.
PARAMETERS:         None
RETURNS:            None
CALLS:              None
CALLED BY:          (KS.PC) init_trees
*****************************************************************************/
{
   c_tree = NULL;
}

c_store(t, info)
struct c_node *t;
struct column info;
/*****************************************************************************
DESCRIPTION:        Puts the contents of info into the info
                    field of the c_node pointer t.
PARAMETERS:         struct c_node *t, struct column info
RETURNS:            None
CALLS:              None
CALLED BY:          c_ins
*****************************************************************************/
{
   t->parent = NULL;
   t->lchild = NULL;
   t->rchild = NULL;
   t->info = info;
}

struct c_node *c_ins(t, info)
struct c_node **t;
struct column info;
/*****************************************************************************
DESCRIPTION:        Creates a node in the tree pointed to by t,
                    and stores the contents of info in that
                    node.  If a node already exists with the
                    same key field, a message is printed
                    warning the user.  If there is insufficient
                    memory to allocate the new node, a warning
                    is printed.
PARAMETERS:         struct c_node **t, struct column info
RETURNS:            Pointer to the new node.  If a node already
                    exists with the same key field, a pointer
                    to that node is returned.  If there is
                    insufficient memory, NULL is returned.
CALLS:              c_store, c_tinit, c_ins
```

```
CALLED BY:          (KS.PC) get_column_info
*****************************************************************************/
{
   if (*t == NULL) {
      *t = (struct c_node *) malloc(sizeof(struct c_node));
      if (*t == NULL) {
         printf("(column insert):  Out of memory\n");
         return NULL;
      }
      c_store(*t, info);
      c_tinit(&(*t)->info);
      return *t;
   }
   else {
      cmp = strcmp(info.name, (*t)->info.name);
      if (cmp < 0) return c_ins(&(*t)->lchild, info);
      else
         if (cmp > 0) return c_ins(&(*t)->rchild, info);
         else {
            printf("*** DUPLICATE KEY (%s) ***\n",info.name);
            return *t;
         }
   }
}

c_deltree(t)
struct c_node *t;
/*****************************************************************************
DESCRIPTION:        Deletes the entire tree pointed to by t.
PARAMETERS:         struct c_node *t
RETURNS:            None
CALLS:              c_deltree, c_tdeltree
CALLED BY:          (KS.PC) del_trees
*****************************************************************************/
{
   if (t->lchild != NULL) {
      c_deltree(t->lchild);
      t->lchild = NULL;
   }
   if (t->rchild != NULL) {
      c_deltree(t->rchild);
      t->rchild = NULL;
   }
   c_tdelrest(&t->info, t->info.tbl_head);
   free(t);
}

struct c_node *c_find(t, name)
struct c_node *t;
char *name;
/*****************************************************************************
DESCRIPTION:        Searches the tree pointed to by t for the
                    node containing name.
PARAMETERS:         struct c_node *t, char *name
RETURNS:            Pointer to the node containing name, or
                    NULL if not found
```

```
CALLS:            c_find
CALLED BY:        c_find
*************************************************************************/
{
   if (t == NULL) return NULL;
   cmp = strcmp(name,t->info.name);
   if (cmp < 0) return c_find(t->lchild, name);
   else
      if (cmp > 0) return c_find(t->rchild, name);
      else return t;
}

struct c_node *c_left(t)
struct c_node **t;
/*************************************************************************
DESCRIPTION:      Changes t to point to its own left child.
                  If t is NULL upon calling the function,
                  nothing is changed.
PARAMETERS:       struct c_node **t
RETURNS:          Returns a pointer to t's left child.  If t
                  is NULL, NULL is returned.
CALLS:            None
CALLED BY:        None
*************************************************************************/
{
   if (*t != NULL) *t = (*t)->lchild;
   return *t;
}

struct c_node *c_right(t)
struct c_node **t;
/*************************************************************************
DESCRIPTION:      Changes t to point to its own right child.
                  If t is NULL upon calling the function,
                  nothing is changed.
PARAMETERS:       struct node **t
RETURNS:          Returns a pointer to t's right child.  If t
                  is NULL, NULL is returned.
CALLS:            None
CALLED BY:        None
*************************************************************************/
{
   if (*t != NULL) *t = (*t)->rchild;
   return *t;
}

c_lnr(t)
struct c_node *t;
/*************************************************************************
DESCRIPTION:      Performs an Inorder traversal (LNR) of the
                  tree pointed to by t while displaying the
                  contents of each node as it is visited.
PARAMETERS:       struct c_node *t
RETURNS:          None
CALLS:            c_lnr, c_shownode
CALLED BY:        c_lnr
```

```
*****************************************************************************/
{
   if (t->lchild != NULL) c_lnr(t->lchild);
   c_shownode(t->info);
   if (t->rchild != NULL) c_lnr(t->rchild);
}

/* - - - - - - - - - - - - - - * * * * * * * * * * - - - - - - - - - - - - - - */
/* - - - - - - - - - - - - - -*   Table List    * - - - - - - - - - - - - - - */
/* - - - - - - - - - - - - - -*    Functions    * - - - - - - - - - - - - - - */
/* - - - - - - - - - - - - - - * * * * * * * * * * - - - - - - - - - - - - - - */
c_tinit(c)
struct column *c;
/*****************************************************************************
DESCRIPTION:      Initializes the table list pointers:
                  tbl_cur, tbl_head, and tbl_tail.
PARAMETERS:       struct column *c
RETURNS:          None
CALLS:            None
CALLED BY:        c_ins
*****************************************************************************/
{
   c->tbl_cur = c->tbl_head = c->tbl_tail = NULL;
}

c_tins(c, info)
struct column *c;
struct c_tab_info info;
/*****************************************************************************
DESCRIPTION:      Creates a new node at the end of the table
                  list and stores the contents of info in the
                  node.  If there is insufficient memory to
                  create the node, a message is printed.
PARAMETERS:       struct column *c, struct c_tab_info info
RETURNS:          None
CALLS:            c_iinit
CALLED BY:        (KS.PC) get_column_info
*****************************************************************************/
{
   if (c->tbl_tail != NULL) {
      c->tbl_tail->next = (struct c_tnode *) malloc(sizeof(struct c_tnode));
      if (c->tbl_tail->next == NULL) {
         printf("(column/table list insert):  Out of memory\n");
         return NULL;
      }
      c->tbl_tail->next->prev = c->tbl_tail;
      c->tbl_tail = c->tbl_tail->next;
      c->tbl_tail->info = info;
      c->tbl_tail->next = NULL;
   }
   else {
      c->tbl_tail = (struct c_tnode *) malloc(sizeof(struct c_tnode));
      if (c->tbl_tail == NULL) {
         printf("(column/table list insert):  Out of memory\n");
         return NULL;
      }
```

```
        c->tbl_head = c->tbl_tail;
        c->tbl_tail->next = NULL;
        c->tbl_head->prev = NULL;
        c->tbl_tail->info = info;
    }
    c_iinit(&c->tbl_tail->info);
}

c_tdel(c, node)
struct column *c;
struct c_tnode *node;
/*****************************************************************************
DESCRIPTION:        Deletes node, pointed to by node, from the
                    table list.
PARAMETERS:         struct column *c, struct c_tnode *node
RETURNS:            None
CALLS:              c_idelrest
CALLED BY:          c_tdelrest
*****************************************************************************/
{
    if (node == NULL) return;
    if ((node == c->tbl_head) || (node == c->tbl_tail)) {
        if (node == c->tbl_head) {
            c->tbl_head = node->next;
            if (c->tbl_head != NULL) c->tbl_head->prev = NULL;
        }
        if (node == c->tbl_tail) {
            c->tbl_tail = node->prev;
            if (c->tbl_tail != NULL) c->tbl_tail->next = NULL;
        }
    }
    else {
        node->prev->next = node->next;
        node->next->prev = node->prev;
    }
    c_idelrest(&node->info, node->info.idx_head);
    free(node);
}

c_tdelrest(c, node)
struct column *c;
struct c_tnode *node;
/*****************************************************************************
DESCRIPTION:        Deletes all nodes from node to tbl_tail
                    (current node to end of list) from the
                    table list.
PARAMETERS:         struct column *c, struct c_tnode *node
RETURNS:            None
CALLS:              c_tdel
CALLED BY:          c_deltree
*****************************************************************************/
{
    struct c_tnode *tmp;

    while (node != NULL) {
        tmp = node->next;
```

```
        c_tdel(c, node);
        node = tmp;
    }
}

struct c_tnode *c_tfind(c, name)
struct column *c;
char *name;
/*****************************************************************************
DESCRIPTION:        Searches the table list for a node
                    containing name.
PARAMETERS:         struct column *c, char *name
RETURNS:            Pointer to the next node
CALLS:              None
CALLED BY:          None
*****************************************************************************/
{
    struct c_tnode *tmp;
}

struct c_tnode *c_tnext(c)
struct column *c;
/*****************************************************************************
DESCRIPTION:        Advances the tbl_cur pointer in c to the
                    next node in the table list.
PARAMETERS:         struct column *c
RETURNS:            Pointer to the next node
CALLS:              None
CALLED BY:          show_column_info
*****************************************************************************/
{
    if (c->tbl_cur != NULL) c->tbl_cur = c->tbl_cur->next;
}

struct c_tnode *c_tprev(c)
struct column *c;
/*****************************************************************************
DESCRIPTION:        Move the tbl_cur pointer in c to the
                    previous node in the table list.
PARAMETERS:         struct column *c
RETURNS:            Pointer to the previous node
CALLS:              None
CALLED BY:          None
*****************************************************************************/
{
    if (c->tbl_cur != NULL) c->tbl_cur = c->tbl_cur->prev;
}

/* - - - - - - - - - - - - - - * * * * * * * * * * - - - - - - - - - - - - - - */
/* - - - - - - - - - - - - - -*    Index List    * - - - - - - - - - - - - - */
/* - - - - - - - - - - - - - -*     Functions    * - - - - - - - - - - - - - */
/* - - - - - - - - - - - - - - * * * * * * * * * * - - - - - - - - - - - - - - */
c_iinit(c)
struct c_tab_info *c;
/*****************************************************************************
DESCRIPTION:        Initializes the index list pointers stored
```

```
                    in c.
PARAMETERS:         struct c_tab_info *c
RETURNS:            None
CALLS:              None
CALLED BY:          c_tins
*****************************************************************************/
{
   c->idx_cur = c->idx_head = c->idx_tail = NULL;
}

c_iins(c, info)
struct c_tab_info *c;
struct c_idx_info info;
/*****************************************************************************
DESCRIPTION:        Creates a new node in the index list and
                    stores the contents of info in that node.
                    If there is insufficient memory to create
                    the new node, a message is printed.
PARAMETERS:         struct c_tab_info *c, struct c_idx_info
                    info
RETURNS:            None
CALLS:              None
CALLED BY:          (KS.PC) get_column_info
*****************************************************************************/
{
   if (c->idx_tail != NULL) {
      c->idx_tail->next = (struct c_inode *) malloc(sizeof(struct c_inode));
      if (c->idx_tail->next == NULL) {
         printf("(column/table/index list insert):  Out of memory\n");
         return NULL;
      }
      c->idx_tail->next->prev = c->idx_tail;
      c->idx_tail = c->idx_tail->next;
      c->idx_tail->info = info;
      c->idx_tail->next = NULL;
   }
   else {
      c->idx_tail = (struct c_inode *) malloc(sizeof(struct c_inode));
      if (c->idx_tail == NULL) {
         printf("(column/table/index list insert):  Out of memory\n");
         return NULL;
      }
      c->idx_head = c->idx_tail;
      c->idx_tail->next = NULL;
      c->idx_head->prev = NULL;
      c->idx_tail->info = info;
   }
}

c_idel(c, node)
struct c_tab_info *c;
struct c_inode *node;
/*****************************************************************************
DESCRIPTION:        Deletes the node, pointed to by node, from
                    the index list.
PARAMETERS:         struct c_tab_info *c, struct c_inode *node
```

```
RETURNS:            None
CALLS:              None
CALLED BY:          c_idelrest
*****************************************************************************/
{
   if (node == NULL) return;
   if ((node == c->idx_head) || (node == c->idx_tail)) {
      if (node == c->idx_head) {
         c->idx_head = node->next;
         if (c->idx_head != NULL) c->idx_head->prev = NULL;
      }
      if (node == c->idx_tail) {
         c->idx_tail = node->prev;
         if (c->idx_tail != NULL) c->idx_tail->next = NULL;
      }
   }
   else {
      node->prev->next = node->next;
      node->next->prev = node->prev;
   }
   free(node);
}

c_idelrest(c, node)
struct c_tab_info *c;
struct c_inode *node;
/*****************************************************************************
DESCRIPTION:        Deletes all nodes from the index list that
                    come after and including the node pointed
                    to by node.
PARAMETERS:         struct c_tab_info *c, struct c_inode *node
RETURNS:            None
CALLS:              c_idel
CALLED BY:          c_tdel
*****************************************************************************/
{
   struct c_inode *tmp;

   while (node != NULL) {
      tmp = node->next;
      c_idel(c, node);
      node = tmp;
   }
}

struct c_inode *c_ifind(c, name)
struct c_tab_info *c;
char *name;
/*****************************************************************************
DESCRIPTION:        Searches the index list for a node
                    containing name.
PARAMETERS:         struct c_tab_info *c, char *name
RETURNS:            Pointer to the next node
CALLS:              None
CALLED BY:          None
*****************************************************************************/
```

```
{
   struct c_inode *tmp;

}

struct c_inode *c_inext(c)
struct c_tab_info *c;
/*****************************************************************************
DESCRIPTION:      Advances the idx_cur pointer stored in c to
                  the next node in the index list.
PARAMETERS:       struct c_tab_info *c
RETURNS:          Pointer to the next node in the index list.
CALLS:            None
CALLED BY:        show_column_info
*****************************************************************************/
{
   if (c->idx_cur != NULL) c->idx_cur = c->idx_cur->next;
}

struct c_inode *c_iprev(c)
struct c_tab_info *c;
/*****************************************************************************
DESCRIPTION:      Changes the idx_cur pointer stored in c to
                  the previous node in the index list.
PARAMETERS:       struct c_tab_info *c
RETURNS:          Pointer to the previous node in the index
                  list.
CALLS:            None
CALLED BY:        None
*****************************************************************************/
{
   if (c->idx_cur != NULL) c->idx_cur = c->idx_cur->prev;
}
```

Appendix D C Code

Oct 11 17:46 1992 i_tree.c Page 1

```
/*****************************************************************************
                                    PROGRAM I_TREE.C
   Project specifications developed by:
         Peggy Wright in support of IAMS research
   Code written by:
         Shannon Thornton           03 AUG 92

   Contains the necessary routines to create, manipulate, and dispose of a
   binary tree.  Each node in the tree contains an index name, table name,
   unique flag, and a linked list of columns.  The list contains all column
   names used by the index.  See TYPES.H for structure definitions.

                                Naming Convention
   Functions used for manipulating the tree or list structure begin with the
   first letter of the structure used.  For example, to insert a column into
   the column tree, c_ins(..) is called.  The c_ indicates the function uses
   the column tree structure.  All functions that manipulate the table list
   inside a node in the column tree begin with c_t.  Any function that
   manipulates the index list inside the previously mentioned table list
   begins with c_i.  All table tree functions start with t_, and all index
   tree functions start with i_.  This convention is also used for global
   tree variables, c_tree, t_tree, and i_tree, which reference the column
   tree, table tree, and index tree, respectively.
*****************************************************************************/

#include <string.h>
#include <stdio.h>
#include <malloc.h>
#include "types.h"

/* - - - - - - - - - - - - - * Global Tree Variables * - - - - - - - - - - - - - */
struct i_node *i_tree;      /* Pointer to the index tree                        */
int cmp;                    /* Temporary variable used in find and insert       */
                            /* Since find and insert are recursive procedures */
                            /* the temporary variable is declared global.  If */
                            /* it was declared locally, it would be           */
                            /* re-allocated on the stack for every function   */
                            /* call.                                          */


/* - - - - - - - - - - - - - * Function definitions * - - - - - - - - - - - - -  */

/* - - - - - - - - - - - - - - - * * * * * * * * * * - - - - - - - - - - - - - - - */
/* - - - - - - - - - - - - - - - * Tree  Functions * - - - - - - - - - - - - - - - */
/* - - - - - - - - - - - - - - - * * * * * * * * * * - - - - - - - - - - - - - - - */
i_shownode(info)
struct index info;
/*****************************************************************************
DESCRIPTION:      Prints the contents of info to standard
                  output.
PARAMETERS:       struct index info
RETURNS:          None
CALLS:            None
CALLED BY:        i_lnr
*****************************************************************************/
{
```

```
    printf("%-32s  %-32s\n",info.name, info.tbl_name);
}

 i_init()
/*****************************************************************************
DESCRIPTION:      Initializes the global tree variable i_tree
                  to NULL.
PARAMETERS:       None
RETURNS:          None
CALLS:            None
CALLED BY:        (KS.PC) init_trees
*****************************************************************************/
{
   i_tree = NULL;
}

i_store(t, info)
struct i_node *t;
struct index info;
/*****************************************************************************
DESCRIPTION:      Puts the contents of info into the info
                  field of the i_node pointer t.
PARAMETERS:       struct i_node *t, struct index info
RETURNS:          None
CALLS:            None
CALLED BY:        i_ins
*****************************************************************************/
{
   t->parent = NULL;
   t->lchild = NULL;
   t->rchild = NULL;
   t->info = info;
}

struct i_node *i_ins(t, info)
struct i_node **t;
struct index info;
/*****************************************************************************
DESCRIPTION:      Creates a node in the tree pointed to by t,
                  and stores the contents of info in that
                  node.  If a node already exists with the
                  same key field, a message is printed
                  warning the user.  If there is insufficient
                  memory to allocate the new node, a warning
                  is printed.
PARAMETERS:       struct i_node **t, struct index info
RETURNS:          Pointer to a new node containing info.  If
                  a node already exists with the same key
                  field, a pointer to that node is returned.
                  If there is insufficient memory, NULL is
                  returned.
CALLS:            i_ins, i_store, i_cinit
CALLED BY:        (KS.PC) get_index_info, i_ins
*****************************************************************************/
{
   if (*t == NULL) {
```

```
        *t = (struct i_node *) malloc(sizeof(struct i_node));
        if (*t == NULL) {
           printf("(index insert): Out of memory\n");
           return NULL;
        }
        i_store(*t, info);
        i_cinit(&(*t)->info);
        return *t;
     }
     else {
        cmp = strcmp(info.name, (*t)->info.name);
        if (cmp < 0) return i_ins(&(*t)->lchild, info);
        else
           if (cmp > 0) return i_ins(&(*t)->rchild, info);
           else {
              printf("*** DUPLICATE KEY (%s) ***\n",info.name);
              return *t;
           }
     }
}

i_deltree(t)
struct i_node *t;
/*****************************************************************************
DESCRIPTION:      Deletes the tree pointed to by t.
PARAMETERS:       struct i_node *t
RETURNS:          None
CALLS:            i_deltree, i_cdelrest
CALLED BY:        (KS.PC) del_trees
*****************************************************************************/
{
   if (t->lchild != NULL) {
      i_deltree(t->lchild);
      t->lchild = NULL;
   }
   if (t->rchild != NULL) {
      i_deltree(t->rchild);
      t->rchild = NULL;
   }
   i_cdelrest(&t->info, t->info.col_head);
   free(t);
}

struct i_node *i_find(t, name)
struct i_node *t;
char *name;
/*****************************************************************************
DESCRIPTION:      Searches the tree pointed to by t for the
                  node containing name.
PARAMETERS:       struct i_node *t, char *name
RETURNS:          Pointer to the node containing name, or
                  NULL if not found
CALLS:            i_find
CALLED BY:        i_find
*****************************************************************************/
{
```

```
    if (t == NULL) return NULL;
   cmp = strcmp(name,t->info.name);
   if (cmp < 0) return i_find(t->lchild, name);
   else
      if (cmp > 0) return i_find(t->rchild, name);
      else return t;
}

struct i_node *i_left(t)
struct i_node **t;
/*****************************************************************************
DESCRIPTION:      Changes t to point to its own left child.
                  If t is NULL upon calling the function,
                  nothing is changed.
PARAMETERS:       struct i_node **t
RETURNS:          Returns a pointer to t's left child.  If t
                  is NULL, NULL is returned.
CALLS:            None
CALLED BY:        None
*****************************************************************************/
{
   if (*t != NULL) *t = (*t)->lchild;
   return *t;
}

struct i_node *i_right(t)
struct i_node **t;
/*****************************************************************************
DESCRIPTION:      Changes t to point to its own right child.
                  If t is NULL upon calling the function,
                  nothing is changed.
PARAMETERS:       struct i_node **t
RETURNS:          Returns a pointer to t's right child.  If t
                  is NULL, NULL is returned.
CALLS:            None
CALLED BY:        None
*****************************************************************************/
{
   if (*t != NULL) *t = (*t)->rchild;
   return *t;
}

 i_lnr(t)
struct i_node *t;
/*****************************************************************************
DESCRIPTION:      Performs an Inorder traversal (LNR) of the
                  tree pointed to by t while displaying the
                  contents of each node as it is visited.
PARAMETERS:       struct i_node *t
RETURNS:          None
CALLS:            i_lnr, i_shownode
CALLED BY:        i_lnr
*****************************************************************************/
{
   if (t->lchild != NULL) i_lnr(t->lchild);
   i_shownode(t->info);
```

```
    if (t->rchild != NULL) i_lnr(t->rchild);
}

/* - - - - - - - - - - - - - - * * * * * * * * * * - - - - - - - - - - - - - - */
/* - - - - - - - - - - - - - - *   index List    * - - - - - - - - - - - - - - */
/* - - - - - - - - - - - - - - *    Functions    * - - - - - - - - - - - - - - */
/* - - - - - - - - - - - - - - * * * * * * * * * * - - - - - - - - - - - - - - */
i_cinit(i)
struct index *i;
/*****************************************************************************
DESCRIPTION:      Initializes the column list pointers:
                  col_cur, col_head, and col_tail.
PARAMETERS:       struct index *i
RETURNS:          None
CALLS:            None
CALLED BY:        i_ins
*****************************************************************************/
{
   i->col_cur = i->col_head = i->col_tail = NULL;
}

i_cins(i, info)
struct index *i;
struct i_col_info info;
/*****************************************************************************
DESCRIPTION:      Creates a new node at the end of the column
                  list and stores the contents of info in the
                  node.  If there is insufficient memory to
                  create the node, a message is printed.
PARAMETERS:       struct index *i, struct i_col_info info
RETURNS:          None
CALLS:            None
CALLED BY:        (KS.PC) get_index_info
*****************************************************************************/
{
   if (i->col_tail != NULL) {
      i->col_tail->next = (struct i_cnode *) malloc(sizeof(struct i_cnode));
      if (i->col_tail->next == NULL) {
         printf("(index/column list insert):  Out of memory\n");
         return NULL;
      }
      i->col_tail->next->prev = i->col_tail;
      i->col_tail = i->col_tail->next;
      i->col_tail->info = info;
      i->col_tail->next = NULL;
   }
   else {
      i->col_tail = (struct i_cnode *) malloc(sizeof(struct i_cnode));
      if (i->col_tail == NULL) {
         printf("(index/column list insert):  Out of memory\n");
         return NULL;
      }
      i->col_head = i->col_tail;
      i->col_tail->next = NULL;
      i->col_head->prev = NULL;
      i->col_tail->info = info;
```

```
    }
}

i_cdel(i, node)
struct index *i;
struct i_cnode *node;
/*****************************************************************************
DESCRIPTION:      Deletes node from the column list.
PARAMETERS:       struct index *i, struct i_cnode *node
RETURNS:          None
CALLS:            None
CALLED BY:        i_cdelrest
*****************************************************************************/
{
   if (node == NULL) return;
   if ((node == i->col_head) || (node == i->col_tail)) {
      if (node == i->col_head) {
         i->col_head = node->next;
         if (i->col_head != NULL) i->col_head->prev = NULL;
      }
      if (node == i->col_tail) {
         i->col_tail = node->prev;
         if (i->col_tail != NULL) i->col_tail->next = NULL;
      }
   }
   else {
      node->prev->next = node->next;
      node->next->prev = node->prev;
   }
   free(node);
}

i_cdelrest(i, node)
struct index *i;
struct i_cnode *node;
/*****************************************************************************
DESCRIPTION:      Deletes all nodes from node to col_tail
                  (current node to end of list) from the
                  column list.
PARAMETERS:       struct index *i, struct i_cnode *node
RETURNS:          None
CALLS:            i_cdel
CALLED BY:        i_deltree
*****************************************************************************/
{
   struct i_cnode *tmp;

   while (node != NULL) {
      tmp = node->next;
      i_cdel(i, node);
      node = tmp;
   }
}

struct i_cnode *i_cfind(i, name)
struct index *i;
```

```
char name[33];
/*****************************************************************************
DESCRIPTION:      Searches the column list for the node
                  containing name.
PARAMETERS:       struct index *i, char *name
RETURNS:          Pointer to a node containing name
CALLS:            None
CALLED BY:        None
*****************************************************************************/
{
   struct i_cnode *imp;
}

struct i_cnode *i_cnext(i)
struct index *i;
/*****************************************************************************
DESCRIPTION:      Advances the col_cur pointer in i to the
                  next node in the column list.
PARAMETERS:       struct index *i
RETURNS:          Pointer to the next node
CALLS:            None
CALLED BY:        (KS.PC) show_index_info
*****************************************************************************/
{
   if (i->col_cur != NULL) i->col_cur = i->col_cur->next;
}

struct i_cnode *i_cprev(i)
struct index *i;
/*****************************************************************************
DESCRIPTION:      Moves the col_cur pointer in i to the
                  previous node in the column list.
PARAMETERS:       struct index *i
RETURNS:          Pointer to the previous node
CALLS:            None
CALLED BY:        None
*****************************************************************************/
{
   if (i->col_cur != NULL) i->col_cur = i->col_cur->prev;
}
```

Appendix D C Code

Oct 11 17:45 1992 iams_const.h Page 1

```
#define NO_DATA      -1
#define NOT_FOUND     0
#define TBL           1
#define TBL_COL_TBL   2
#define MULTITBL      3
#define MULTIJC       4

#define OK            0
#define JCLIST        1
#define NOTREL        2
#define OKNOROWS      3
#define TABLELIST     4

/* Priorities */
#define LEVEL6        6
#define LEVEL5        5
#define LEVEL4        4
#define LEVEL3        3
#define LEVEL2        2
#define LEVEL1        1
#define LEVEL0        0
#define NOPRI        -1

#define max(a, b) ((a > b) ? (a) : (b))
#define min(a, b) ((a < b) ? (a) : (b))

#define NAMELENGTH   31
```

Appendix D C Code

Oct 11 17:44 1992 iui.c Page 1

```
/*****************************************************************************
                                    PROGRAM IUI.C
   IAMS intelligent user interface code utilizes Motif functions to accept
      user input from the screen, and display system output to the screen.
   Project specifications developed by:
         Peggy Wright in support of IAMS research
   Code written by:
         Shannon Thornton           01 SEP 92
*****************************************************************************/

#include <stdio.h>
#include <string.h>
#include <Xm/Xm.h>
#include <Xm/PushB.h>
#include <Xm/Label.h>
#include <Xm/Separator.h>
#include <Xm/Form.h>
#include <Xm/Text.h>
#include <Xm/MessageB.h>
#include <Xm/List.h>
#include "mks.h"

extern struct m_node *m_tree;
FILE *iams_log;

/*********/
/* Misc. */
/*********/
XtAppContext context;
XmStringCharSet char_set=XmSTRING_DEFAULT_CHARSET;

#define NUMOPS 7

/* SQL Constants */
#define NOT_RESERVED -1
#define NOTEQ1 0
#define LTE    1
#define NOTEQ2 2
#define EQ     3
#define GTE    4
#define AND    5
#define BY     6
#define FROM   7
#define GROUP  8
#define HAVING 9
#define IN     10
#define LIKE   11
#define NOT    12
#define OR     13
#define ORDER  14
#define SELECT 15
#define WHERE  16
#define NUM_RESERVED 18

char *ops[]={"=","<","<=",">",">=","<>","LIKE"};
```

Oct 11 17:44 1992 iui.c Page 2

```
char reserved_word[] [NUM_RESERVED] =
    {"!=","<=","<>","=",">=","AND","BY","FROM","GROUP","HAVING","IN","LIKE",
     "NOT","OR","ORDER","SELECT","WHERE",NULL};

#define TABLENAME 1
#define JCNAME    2
#define APPNAME   3

int fid, listtype;
char t1namestr[30], t2namestr[30], jcnamestr[30], t1opstr[5], t2opstr[5],
   t1rangestr[30], t2rangestr[30], app[30], app1[30], app2[30];

/***********/
/* Widgets */
/***********/
Widget toplevel, form, quit, execute, sclabel, name, operator, range,
   target1, target2, t1name, t1op, t1range, t2name, t2op, t2range,
   sep, sep2, errdlg, msgdlg, wrkdlg, qry, res, jclst, uiddlg,
   qrylabel, reslabel, jclabel;


void Init()
{
   iams_log = fopen("iams_log.txt", "wt");
   mksstart();
}

void Quit()
{
   mksquit();
   fclose(iams_log);
}

/**********************/
/* Utility functions */
/**********************/
void set_params(app_name, 11, 12, 13, 14, 15, 16)
char *app_name, *11, *12, *13, *14, *15, *16;
{
   FILE *f;
   char fname[50];

   sprintf(fname,"iams_%s_results.txt",app_name);
   strlwr(fname);
   if ((f = fopen(fname, "wt")) == NULL) {
      fprintf(iams_log, "IUI:  Unable to create %s\n",fname);
      Quit();
      exit(-1);
   }
   fprintf(f,"%s\n%s\n%s\n%s\n%s\n%s\n",11,12,13,14,15,16);
   fclose(f);
}

void msgCB(w, client_data, call_data)
Widget w;
int client_data;
```

```
XmSelectionBoxCallbackStruct *call_data;
{
   XtUnmanageChild(w);
}

void errwin(msg)
char *msg;
{
   Arg al[10];
   int ac;

   ac=0;
   XtSetArg(al[ac], XmNmessageString, XmStringCreateLtoR(msg,char_set));
   ac++;
   XtSetValues(errdlg,al,ac);
   XtManageChild(errdlg);
}

void msgwin(msg)
char *msg;
{
   Arg al[10];
   int ac;
   Widget win;

   ac=0;
   XtSetArg(al[ac], XmNmessageString, XmStringCreateLtoR(msg,char_set));
   ac++;
   XtSetValues(msgdlg,al,ac);
   XtManageChild(msgdlg);
}

void wrkwin(msg)
char *msg;
{
   Arg al[10];
   int ac;
   Widget win;

   ac=0;
   XtSetArg(al[ac], XmNmessageString, XmStringCreateLtoR(msg,char_set));
   ac++;
   XtSetValues(wrkdlg,al,ac);
   XtManageChild(wrkdlg);
}

void removewrkwin()
{
   XtUnmanageChild(wrkdlg);
}

int is_reserved(w)
char *w;
{
   int i = 0, cmp = 1;
```

```
   while ((reserved_word[i] != NULL) &&
          ((cmp = strcmp(reserved_word[i],w)) < 0))
      i++;
   if ((cmp == 0) && (reserved_word[i] != NULL)) return i;
   else return -1;
}

char *format_sql(s)
char *s;
{
   char t[4096], *word, delta[10];
   int res;

   t[0] = delta[0] = '\0';
   word = (char *) strtok(s, " ");
   while (word != NULL) {
      if ((res = is_reserved(word)) == NOT_RESERVED)
         sprintf(t, "%s %s", t, word);
      else
         switch (res) {
            case NOTEQ1:
            case LTE:
            case NOTEQ2:
            case EQ:
            case GTE:
            case LIKE:
               sprintf(t,"%s %s", t, word);
               break;
            case BY:
               sprintf(t,"%s %s", t, word);
               break;
            case IN:
               sprintf(t,"%s %s\n", t, word);
               strcat(delta, "\t");
               break;
            case GROUP:
            case ORDER:
               sprintf(t,"%s\n%s%s", t, delta, word);
               break;
            case AND:
            case FROM:
            case HAVING:
            case OR:
            case WHERE:
               sprintf(t,"%s\n%s%-*s", t, delta, 8, word);
               break;
            case SELECT:
               sprintf(t,"%s%s%-*s", t, delta, 8, word);
               break;
         }
      word = (char *) strtok(NULL, " ");
   }
   strcpy(s, t);
   return s;
}
```

Oct 11 17:44 1992 iui.c Page 5

```
Boolean get_query()
{
   FILE *f;
   char query[2048];
   int tp;

   printf("Entering get_query\n");
   if ((f = fopen("iams_qry.txt", "rt")) == NULL) {
      errwin("Unable to open iams_qry.txt!");
      return False;
   }
   fgets(query, 2048, f);
   query[strlen(query)-1] = '\0';
   printf("Read:  %s\n",query);
   format_sql(query);
   printf("Formatted:\n%s\n",query);
   query[strlen(query)+1] = '\0';
   query[strlen(query)] = '\n';
   tp = XmTextGetLastPosition(qry);
   XmTextInsert(qry, (XmTextPosition) tp, query);
   tp += strlen(query);
   strcpy(query,"\n\n");
   XmTextInsert(qry, (XmTextPosition) tp, query);
   tp += 2;
   fclose(f);
   unlink("iams_qry.txt");
   return True;
}

Boolean get_results(app)
char *app;
{
   char fname[80], line[128], rescode[5];
   FILE *f;
   int tp, irescode;
   XmString s;
   Boolean retval = False;

   sprintf(fname, "iams_%s_results.txt", app);
   strlwr(fname);
   if ((f = fopen(fname, "rt")) == NULL) {
      sprintf(line,"Unable to open %s!",fname);
      errwin(line);
      return False;
   }
   fgets(rescode, 5, f);
   irescode = rescode[0] - 48;
   switch (irescode) {
      case OK:
         fgets(line, 128, f);
         tp = XmTextGetLastPosition(res);
         while (feof(f) == 0) {
            printf("RESULTS:  %s", line);
            XmTextInsert(res, (XmTextPosition) tp, line);
            tp += strlen(line);
            fgets(line, 128, f);
```

```
        }
          strcpy(line,"\n\n");
          XmTextInsert(res, (XmTextPosition) tp, line);
          tp += 2;
          retval = True;
          break;
       case NOTREL:
          msgwin("A relationship could not be found between T1 and T2.");
          retval = False;
          break;
       case OKNOROWS:
          msgwin("No data was found matching the specifications.");
          unlink("iams_qry.txt");
          retval = False;
          break;
    }
    fclose(f);
    unlink(fname);
    return retval;
}

/*****************/
/* MKS Searches */
/*****************/
Boolean single_app(t1, t2)
struct m_node *t1, *t2;
{
    if (t1 == NULL) {
       if (t2->info.app_head != NULL) {
          strcpy(app,t2->info.app_head->info.name);
          return True;
       }
    }
    else if (t2 == NULL) {
       if (t1->info.app_head != NULL) {
          strcpy(app,t1->info.app_head->info.name);
          return True;
       }
    }
    else { /* Both are NOT NULL */
    t1->info.app_cur = t1->info.app_head;
    while (t1->info.app_cur != NULL) {
       t2->info.app_cur = t2->info.app_head;
       while (t2->info.app_cur != NULL) {
          if (strcmp(t1->info.app_cur->info.name,
             t2->info.app_cur->info.name) == 0) {
             strcpy(app,t1->info.app_cur->info.name);
             return True;
          }
          m_anext(&t2->info);
       }
       m_anext(&t1->info);
    }
    } /* End Both are NOT NULL */
    return False;
}
```

Oct 11 17:44 1992 iui.c Page 7

```
void multi_app(t1, t2, m, count, app1, app2)
struct m_node *t1, *t2, *m;
int *count;
char *app1, *app2;
{
   XmString s;
   struct m_anode *tmp;

   if (m == NULL) return;

/* Search for a relation (join) column across databases/applications/KS */
   if ((strcmp(m->info.name,t1->info.name) != 0) &&
       (strcmp(m->info.name,t2->info.name) != 0)) {
      m->info.app_cur = m->info.app_head;
      while (m->info.app_cur != NULL) {
         t1->info.app_cur = t1->info.app_head;
         while (t1->info.app_cur != NULL) {
            if (strcmp(m->info.app_cur->info.name,
                t1->info.app_cur->info.name) == 0) {
               tmp = m->info.app_cur;
               m->info.app_cur = m->info.app_head;
               while (m->info.app_cur != NULL) {
                  t2->info.app_cur = t2->info.app_head;
                  while (t2->info.app_cur != NULL) {
                     if (strcmp(m->info.app_cur->info.name,
                        t2->info.app_cur->info.name) == 0) {
                           (*count)++;
                           strcpy(app1,tmp->info.name);
                           strcpy(app2,m->info.app_cur->info.name);
                           strcpy(jcnamestr,m->info.name);
                           printf("\t\tPotential JC(%d):  %s\n",
                              *count,jcnamestr);
                           s=XmStringCreate(jcnamestr, char_set);
                           XmListAddItem(jclst,s,0);
                           XmStringFree(s);
                     }
                     m_anext(&t2->info);
                  }
                  m_anext(&m->info);
               }
            }
            m_anext(&t1->info);
         }
         m_anext(&m->info);
      }
   }
   multi_app(t1, t2, m->lchild, count, app1, app2);
   multi_app(t1, t2, m->rchild, count, app1, app2);
}

/*************/
/* Callbacks */
/*************/
void selectCB(w, client_data, call_data)
Widget w;
```

```
int client_data;
XmListCallbackStruct *call_data;
{
   char *item;

   XmStringGetLtoR(call_data->item,char_set,&item);
   XtUr anageChild(jclst);
   XtUnmanageChild(jclabel);
   XmListDeleteAllItems(jclst);
   printf("\tForking ks for %s. . .\n",app1);
   set_params(app1,t1namestr,t1opstr,t1rangestr,item,"","");
   if ((fid = fork()) == 0) {
      execl("ks", "ks", app1, (char *) 0);
      errwin("Unable to execute knowledge source.");
   }
   else wait((int *) 0);
   printf("\tReading query and results.\n");
   if (get_results(app1))
      get_query();
   else printf("Error reading results for %s\n",app1);
   printf("\tForking ks for %s. . .\n",app2);
   set_params(app2,item,"","",t2namestr,t2opstr,t2rangestr);
   if ((fid = fork()) == 0) {
      execl("ks", "ks", app2, (char *) 0);
      errwin("Unable to execute knowledge source.");
   }
   else wait((int *) 0);
   printf("\tReading query and results.\n");
   if (get_results(app2))
      get_query();
   else printf("Error reading results for %s\n",app2);
   XtFree(item);
}

void quitCB(w, client_data, call_data)
Widget w;
int client_data;
XmPushButtonCallbackStruct *call_data;
{
   Quit();
   exit(0);
}

void executeCB(w, client_data, call_data)
Widget w;
int client_data;
XmPushButtonCallbackStruct *call_data;
{
   int i, ac, numjcs;
   Arg al[5];
   struct m_node *t1 = NULL, *t2 = NULL;
   Boolean found;

   strcpy(t1namestr, XmTextGetString(t1name));
   strcpy(t2namestr, XmTextGetString(t2name));
   strcpy(t1opstr, XmTextGetString(t1op));
```

```
    strcpy(t2opstr, XmTextGetString(t2op));
   strcpy(t1rangestr, XmTextGetString(t1range));
   strcpy(t2rangestr, XmTextGetString(t2range));
   trim(t1namestr);
   trim(t2namestr);
   trim(t1opstr);
   trim(t2opstr);
   trim(t1rangestr);
   trim(t2rangestr);
   strupr(t1namestr);
   strupr(t2namestr);

   printf("Target 1:  %30s %5s %30s\n",t1namestr,t1opstr,t1rangestr);
   printf("Target 2:  %30s %5s %30s\n",t2namestr,t2opstr,t2rangestr);

/* Check names to be sure both are not empty */
   if (t1namestr[0] == NULL && t2namestr[0] == NULL) {
      msgwin("Both Target 1 and Target 2 cannot be empty!");
      return;
   }

/* Make sure Target 1 and/or Target 2 exist */
   if (t1namestr[0] != NULL)
      if ((t1 = (struct m_node *) m_find(m_tree, t1namestr)) == NULL) {
         errwin("Target 1 is not a valid column name.");
         return;
      }
   if (t2namestr[0] != NULL)
      if ((t2 = (struct m_node *) m_find(m_tree, t2namestr)) == NULL) {
         errwin("Target 2 is not a valid column name.");
         return;
      }

/* Check for invalid operators */
   if (t1opstr[0] != NULL) {
      strupr(t1opstr);
      for (i=0; i<NUMOPS; i++)
         if (strcmp(t1opstr,ops[i])==0) break;
      if (i >= NUMOPS) {
         errwin("The operator for Target 1 is invalid");
         return;
      }
   }
   if (t2opstr[0] != NULL) {
      strupr(t2opstr);
      for (i=0; i<NUMOPS; i++)
         if (strcmp(t2opstr,ops[i])==0) break;
      if (i >= NUMOPS) {
         errwin("The operator for Target 2 is invalid");
         return;
      }
   }

/* Check for invalid ranges */
/* The only invalid range is a numeric column with a character range */
   if (t1 != NULL)
```

```
        if (t1->info.type[0] == 'N') {
          for (i=0; i<strlen(t1rangestr); i++)
             if (isalpha(t1rangestr[i]))
                break;
          if (i<strlen(t1rangestr)) {
             errwin(
   "Target 1 is a numeric column, but its range contains character data."
             );
             return;
          }
       }
   if (t2 != NULL)
      if (t2->info.type[0] == 'N') {
         for (i=0; i<strlen(t2rangestr); i++)
            if (isalpha(t2rangestr[i]))
               break;
         if (i<strlen(t2rangestr)) {
            errwin(
   "Target 2 is a numeric column, but its range contains character data."
            );
            return;
         }
      }

/* Check for incomplete search constraints */
/* Operator is NON-NULL and range is NULL or vice-versa */
   if ((t1opstr[0] != NULL) && (t1rangestr[0] == NULL)) {
      errwin("The range for Target 1 is blank, but the operator is not.");
      return;
   }
   if ((t1opstr[0] == NULL) && (t1rangestr[0] != NULL)) {
      errwin("The operator for Target 1 is blank, but the range is not.");
      return;
   }
   if ((t2opstr[0] != NULL) && (t2rangestr[0] == NULL)) {
      errwin("The range for Target 2 is blank, but the operator is not.");
      return;
   }
   if ((t2opstr[0] == NULL) && (t2rangestr[0] != NULL)) {
      errwin("The operator for Target 2 is blank, but the range is not.");
      return;
   }

/* Try to find a common application for T1 & T2 */
   if (single_app(t1, t2)) {
      printf("\tForking ks . . .\n");
      printf("\tks %s\n",app);
      set_params(app,t1namestr,t1opstr,t1rangestr,t2namestr,
         t2opstr,t2rangestr);
      if ((fid = fork()) == 0) {
         execl("ks", "ks", app, (char *) 0);
         errwin("Unable to execute knowledge source.");
      }
      else wait((int *) 0);
      printf("\tReading query and results.\n");
      if (get_results(app))
```

```
            get_query();
         return;
      }
      else {
         printf("Trying multi_app . . .\n");
         numjcs = 0;
         multi_app(t1, t2, m_tree, &numjcs, app1, app2);
         printf("Multi_app found %d join columns\n",numjcs);
         if (numjcs > 1) {
            XtManageChild(jclabel);
            XtManageChild(jclst);
         }
         else
            if (numjcs < 1) {
               msgwin("Target 1 and Target 2 are not related.");
               return;
            }
            else {
               set_params(app1,t1namestr,t1opstr,t1rangestr,jcnamestr,"","");
               if ((fid = fork()) == 0) {
                  execl("ks", "ks", app1, (char *) 0);
                  errwin("Unable to execute knowledge source.");
                  return;
               }
               else wait((int *) 0);
               if (get_results(app1))
                  get_query();
               set_params(app2,jcnamestr,"","",t2namestr,t2opstr,t2rangestr);
               if ((fid = fork()) == 0) {
                  execl("ks", "ks", app2, (char *) 0);
                  errwin("Unable to execute knowledge source.");
                  return;
               }
               else wait((int *) 0);
               if (get_results(app2))
                  get_query();
            }
      }
}

main(argc, argv)
int argc;
char *argv[];
{
   Arg al[10];
   int ac, i;
   XmString s;
   Widget list;

   /* Initialize MKS and log file */
   Init();

   /* create the toplevel shell */
   toplevel = XtAppInitialize(&context, "", NULL, 0, &argc, argv,
      NULL, NULL, 0);
```

```
    /* create message window */
   msgdlg=XmCreateInformationDialog(toplevel,"msgdlg",NULL,0);
   XtAddCallback(msgdlg,XmNokCallback,msgCB,NULL);
   XtUnmanageChild(XmMessageBoxGetChild(msgdlg,
      XmDIALOG_CANCEL_BUTTON));
   XtUnmanageChild(XmMessageBoxGetChild(msgdlg,
      XmDIALOG_HELP_BUTTON));

   /* create error window */
   errdlg=XmCreateErrorDialog(toplevel,"errdlg",NULL,0);
   XtAddCallback(errdlg,XmNokCallback,msgCB,NULL);
   XtUnmanageChild(XmMessageBoxGetChild(errdlg,
      XmDIALOG_CANCEL_BUTTON));
   XtUnmanageChild(XmMessageBoxGetChild(errdlg,
      XmDIALOG_HELP_BUTTON));

   /* create working window */
   wrkdlg=XmCreateInformationDialog(toplevel,"wrkdlg",NULL,0);
   XtAddCallback(wrkdlg,XmNokCallback,msgCB,NULL);
   XtUnmanageChild(XmMessageBoxGetChild(wrkdlg,
      XmDIALOG_CANCEL_BUTTON));
   XtUnmanageChild(XmMessageBoxGetChild(wrkdlg,
      XmDIALOG_HELP_BUTTON));

   /* create a form to hold the widgets */
   ac=0;
   XtSetArg(al[ac],XmNheight,800); ac++;
   XtSetArg(al[ac],XmNwidth,800); ac++;
   form=XmCreateForm(toplevel,"form",al,ac);
   XtManageChild(form);

   /* create the JC list */
   ac=0;
   XtSetArg(al[ac],XmNtopItemPosition,0); ac++;
   XtSetArg(al[ac],XmNvisibleItemCount,10); ac++;
   /*jclst=XmCreateScrolledList(form, "jclst",al,ac);*/
   jclst=XmCreateList(form, "jclst",al,ac);
   XtAddCallback(jclst,XmNdefaultActionCallback,selectCB,NULL);

   /* create the Query text */
   ac=0;
   XtSetArg(al[ac],XmNeditable,False); ac++;
   /*qry=XmCreateScrolledText(form,"qry",al,ac);*/
   qry=XmCreateText(form,"qry",al,ac);
   XtManageChild(qry);

   /* create the Results text */
   ac=0;
   XtSetArg(al[ac],XmNeditable,False); ac++;
   /*res=XmCreateScrolledText(form,"res",al,ac);*/
   res=XmCreateText(form,"res",al,ac);
   XtManageChild(res);

   /* create a label for the Query text */
   ac=0;
   XtSetArg(al[ac],XmNlabelString,
```

Oct 11 17:44 1992 iui.c Page 13

```
      XmStringCreate("Query Executed",char_set)); ac++;
   qrylabel=XmCreateLabel(form,"qrylabel",al,ac);
   XtManageChild(qrylabel);

   /* create a label for the Results text */
   ac=0;
   XtSetArg(al[ac],XmNlabelString,
      XmStringCreate("Results of the Query",char_set)); ac++;
   reslabel=XmCreateLabel(form,"reslabel",al,ac);
   XtManageChild(reslabel);

   /* create a label for the jc list */
   ac=0;
   XtSetArg(al[ac],XmNlabelString,
      XmStringCreateLtoR("Double click the\ndesired join column",char_set));
   ac++;
   jclabel=XmCreateLabel(form,"jclabel1",al,ac);

   /* create EXECUTE button */
   ac=0;
   XtSetArg(al[ac],XmNlabelString,XmStringCreate("Execute",char_set)); ac++;
   execute=XmCreatePushButton(form,"execute",al,ac);
   XtManageChild(execute);
   XtAddCallback(execute,XmNactivateCallback,executeCB,NULL);

   /* create QUIT button */
   ac=0;
   XtSetArg(al[ac],XmNlabelString,XmStringCreate("Quit",char_set)); ac++;
   quit=XmCreatePushButton(form,"quit",al,ac);
   XtManageChild(quit);
   XtAddCallback(quit,XmNactivateCallback,quitCB,NULL);

   /* create "Search Constraints" label */
   ac=0;
   XtSetArg(al[ac],XmNlabelString,
      XmStringCreate("Search Constraints", char_set)); ac++;
   sclabel=XmCreateLabel(form,"sclabel",al,ac);
   XtManageChild(sclabel);

   /* create "Name" label */
   ac=0;
   XtSetArg(al[ac],XmNlabelString,
      XmStringCreate("Name", char_set)); ac++;
   name=XmCreateLabel(form,"name",al,ac);
   XtManageChild(name);

   /* create "Operator" label */
   ac=0;
   XtSetArg(al[ac],XmNlabelString,
      XmStringCreate("Operator", char_set)); ac++;
   operator=XmCreateLabel(form,"operator",al,ac);
   XtManageChild(operator);

   /* create "Range" label */
   ac=0;
   XtSetArg(al[ac],XmNlabelString,
```

```
        XmStringCreate("Range", char_set)); ac++;
    range=XmCreateLabel(form,"range",al,ac);
    XtManageChild(range);

    /* create "Target 1" label */
    ac=0;
    XtSetArg(al[ac],XmNlabelString,
       XmStringCreate("Target 1", char_set)); ac++;
    target1=XmCreateLabel(form,"target1",al,ac);
    XtManageChild(target1);

    /* create "Target 2" label */
    ac=0;
    XtSetArg(al[ac],XmNlabelString,
       XmStringCreate("Target 2", char_set)); ac++;
    target2=XmCreateLabel(form,"target2",al,ac);
    XtManageChild(target2);

    /* create t1name text field */
    ac=0;
    XtSetArg(al[ac],XmNmaxLength,30); ac++;
    t1name=XmCreateText(form,"t1name",al,ac);
    XtManageChild(t1name);

    /* create t1op text field */
    ac=0;
/* This was a Text InputLine */

    XtSetArg(al[ac],XmNmaxLength,5); ac++;
    t1op=XmCreateText(form,"t1op",al,ac);
/*
    XtSetArg(al[ac],XmNautoUnmanage,False); ac++;
    XtSetArg(al[ac],XmNmustMatch,True); ac++;
    XtSetArg(al[ac],XmNselectionLabelString,
       XmStringCreateLtoR("Pick an operator:",char_set)); ac++;
    t1op=XmCreateSelectionDialog(form,"t1op",al,ac);
    list=XmSelectionBoxGetChild(t1op,XmDIALOG_LIST);
    XmListDeleteAllItems(list);
    for (i=0; i<XtNumber(ops); i++) {
       s=XmStringCreate(ops[i],char_set);
       XmListAddItem(list,s,0);
       XmStringFree(s);
    }
    XtUnmanageChild(XmSelectionBoxGetChild(t1op,XmDIALOG_HELP_BUTTON));
*/
    XtManageChild(t1op);

    /* create t1range text field */
    ac=0;
    XtSetArg(al[ac],XmNmaxLength,50); ac++;
    t1range=XmCreateText(form,"t1range",al,ac);
    XtManageChild(t1range);

    /* create t2name text field */
    ac=0;
    XtSetArg(al[ac],XmNmaxLength,30); ac++;
```

```
    t2name=XmCreateText(form,"t2name",al,ac);
   XtManageChild(t2name);

   /* create t2op text field */
   ac=0;
   XtSetArg(al[ac],XmNmaxLength,5); ac++;
   t2op=XmCreateText(form,"t2op",al,ac);
   XtManageChild(t2op);

   /* create t2range text field */
   ac=0;
   XtSetArg(al[ac],XmNmaxLength,50); ac++;
   t2range=XmCreateText(form,"t2range",al,ac);
   XtManageChild(t2range);

   /* create a separator */
   ac=0;
   sep=XmCreateSeparator(form,"sep",al,ac);
   XtManageChild(sep);

   /* create another separator */
   ac=0;
   sep2=XmCreateSeparator(form,"sep2",al,ac);
   XtManageChild(sep2);

   /* attach the children to the form */

   ac=0;
   XtSetArg(al[ac],XmNtopAttachment, XmATTACH_FORM); ac++;
   XtSetArg(al[ac],XmNrightAttachment, XmATTACH_POSITION); ac++;
   XtSetArg(al[ac],XmNrightPosition, 50); ac++;
   XtSetArg(al[ac],XmNleftAttachment, XmATTACH_FORM); ac++;
   XtSetArg(al[ac],XmNbottomAttachment, XmATTACH_POSITION); ac++;
   XtSetArg(al[ac],XmNbottomPosition, 5); ac++;
   XtSetValues(execute,al,ac);

   ac=0;
   XtSetArg(al[ac],XmNtopAttachment, XmATTACH_FORM); ac++;
   XtSetArg(al[ac],XmNrightAttachment, XmATTACH_FORM); ac++;
   XtSetArg(al[ac],XmNleftAttachment, XmATTACH_WIDGET); ac++;
   XtSetArg(al[ac],XmNleftWidget, execute); ac++;
   XtSetArg(al[ac],XmNbottomAttachment, XmATTACH_POSITION); ac++;
   XtSetArg(al[ac],XmNbottomPosition, 5); ac++;
   XtSetValues(quit,al,ac);

   ac=0;
   XtSetArg(al[ac],XmNtopAttachment, XmATTACH_WIDGET); ac++;
   XtSetArg(al[ac],XmNtopWidget, execute); ac++;
   XtSetArg(al[ac],XmNrightAttachment, XmATTACH_FORM); ac++;
   XtSetArg(al[ac],XmNleftAttachment, XmATTACH_FORM); ac++;
   XtSetArg(al[ac],XmNbottomAttachment, XmATTACH_NONE); ac++;
   XtSetValues(sep,al,ac);

   ac=0;
   XtSetArg(al[ac],XmNtopAttachment, XmATTACH_WIDGET); ac++;
   XtSetArg(al[ac],XmNtopWidget, sep); ac++;
```

```
    XtSetArg(al[ac],XmNrightAttachment, XmATTACH_FORM); ac++;
   XtSetArg(al[ac],XmNleftAttachment, XmATTACH_OPPOSITE_WIDGET); ac++;
   XtSetArg(al[ac],XmNleftWidget, operator); ac++;
   XtSetArg(al[ac],XmNbottomAttachment, XmATTACH_WIDGET); ac++;
   XtSetArg(al[ac],XmNbottomWidget, range); ac++;
   XtSetValues(sclabel,al,ac);

   ac=0;
   XtSetArg(al[ac],XmNtopAttachment, XmATTACH_NONE); ac++;
   XtSetArg(al[ac],XmNrightAttachment, XmATTACH_WIDGET); ac++;
   XtSetArg(al[ac],XmNrightWidget, operator); ac++;
   XtSetArg(al[ac],XmNleftAttachment, XmATTACH_OPPOSITE_WIDGET); ac++;
   XtSetArg(al[ac],XmNleftWidget, tlname); ac++;
   XtSetArg(al[ac],XmNbottomAttachment, XmATTACH_WIDGET); ac++;
   XtSetArg(al[ac],XmNbottomWidget, tlname); ac++;
   XtSetValues(name,al,ac);

   ac=0;
   XtSetArg(al[ac],XmNtopAttachment, XmATTACH_NONE); ac++;
   XtSetArg(al[ac],XmNrightAttachment, XmATTACH_OPPOSITE_WIDGET); ac++;
   XtSetArg(al[ac],XmNrightWidget, tlop); ac++;
   XtSetArg(al[ac],XmNleftAttachment, XmATTACH_OPPOSITE_WIDGET); ac++;
   XtSetArg(al[ac],XmNleftWidget, tlop); ac++;
   XtSetArg(al[ac],XmNbottomAttachment, XmATTACH_WIDGET); ac++;
   XtSetArg(al[ac],XmNbottomWidget, tlop); ac++;
   XtSetValues(operator,al,ac);

   ac=0;
   XtSetArg(al[ac],XmNtopAttachment, XmATTACH_NONE); ac++;
   XtSetArg(al[ac],XmNrightAttachment, XmATTACH_FORM); ac++;
   XtSetArg(al[ac],XmNleftAttachment, XmATTACH_OPPOSITE_WIDGET); ac++;
   XtSetArg(al[ac],XmNleftWidget, tlrange); ac++;
   XtSetArg(al[ac],XmNbottomAttachment, XmATTACH_WIDGET); ac++;
   XtSetArg(al[ac],XmNbottomWidget, tlrange); ac++;
   XtSetValues(range,al,ac);

   ac=0;
   XtSetArg(al[ac],XmNtopAttachment, XmATTACH_NONE); ac++;
   XtSetArg(al[ac],XmNrightAttachment, XmATTACH_WIDGET); ac++;
   XtSetArg(al[ac],XmNrightWidget, tlname); ac++;
   XtSetArg(al[ac],XmNleftAttachment, XmATTACH_FORM); ac++;
   XtSetArg(al[ac],XmNbottomAttachment, XmATTACH_POSITION); ac++;
   XtSetArg(al[ac],XmNbottomPosition, 20); ac++;
   XtSetValues(target1,al,ac);

   ac=0;
   XtSetArg(al[ac],XmNtopAttachment, XmATTACH_NONE); ac++;
   XtSetArg(al[ac],XmNrightAttachment, XmATTACH_WIDGET); ac++;
   XtSetArg(al[ac],XmNrightWidget, tlop); ac++;
   XtSetArg(al[ac],XmNleftAttachment, XmATTACH_NONE); ac++;
   XtSetArg(al[ac],XmNbottomAttachment, XmATTACH_POSITION); ac++;
   XtSetArg(al[ac],XmNbottomPosition, 20); ac++;
   XtSetValues(tlname,al,ac);

   ac=0;
   XtSetArg(al[ac],XmNtopAttachment, XmATTACH_NONE); ac++;
```

```
    XtSetArg(al[ac],XmNrightAttachment, XmATTACH_WIDGET); ac++;
    XtSetArg(al[ac],XmNrightWidget, t1range); ac++;
    XtSetArg(al[ac],XmNleftAttachment, XmATTACH_NONE); ac++;
    XtSetArg(al[ac],XmNbottomAttachment, XmATTACH_POSITION); ac++;
    XtSetArg(al[ac],XmNbottomPosition, 20); ac++;
    XtSetValues(t1op,al,ac);

    ac=0;
    XtSetArg(al[ac],XmNtopAttachment, XmATTACH_NONE); ac++;
    XtSetArg(al[ac],XmNrightAttachment, XmATTACH_FORM); ac++;
    XtSetArg(al[ac],XmNleftAttachment, XmATTACH_NONE); ac++;
    XtSetArg(al[ac],XmNbottomAttachment, XmATTACH_POSITION); ac++;
    XtSetArg(al[ac],XmNbottomPosition, 20); ac++;
    XtSetValues(t1range,al,ac);

    ac=0;
    XtSetArg(al[ac],XmNtopAttachment, XmATTACH_NONE); ac++;
    XtSetArg(al[ac],XmNrightAttachment, XmATTACH_WIDGET); ac++;
    XtSetArg(al[ac],XmNrightWidget, t2name); ac++;
    XtSetArg(al[ac],XmNleftAttachment, XmATTACH_FORM); ac++;
    XtSetArg(al[ac],XmNbottomAttachment, XmATTACH_POSITION); ac++;
    XtSetArg(al[ac],XmNbottomPosition, 35); ac++;
    XtSetValues(target2,al,ac);

    ac=0;
    XtSetArg(al[ac],XmNtopAttachment, XmATTACH_NONE); ac++;
    XtSetArg(al[ac],XmNrightAttachment, XmATTACH_WIDGET); ac++;
    XtSetArg(al[ac],XmNrightWidget, t2op); ac++;
    XtSetArg(al[ac],XmNleftAttachment, XmATTACH_NONE); ac++;
    XtSetArg(al[ac],XmNbottomAttachment, XmATTACH_POSITION); ac++;
    XtSetArg(al[ac],XmNbottomPosition, 35); ac++;
    XtSetValues(t2name,al,ac);

    ac=0;
    XtSetArg(al[ac],XmNtopAttachment, XmATTACH_NONE); ac++;
    XtSetArg(al[ac],XmNrightAttachment, XmATTACH_WIDGET); ac++;
    XtSetArg(al[ac],XmNrightWidget, t2range); ac++;
    XtSetArg(al[ac],XmNleftAttachment, XmATTACH_NONE); ac++;
    XtSetArg(al[ac],XmNbottomAttachment, XmATTACH_POSITION); ac++;
    XtSetArg(al[ac],XmNbottomPosition, 35); ac++;
    XtSetValues(t2op,al,ac);

    ac=0;
    XtSetArg(al[ac],XmNtopAttachment, XmATTACH_NONE); ac++;
    XtSetArg(al[ac],XmNrightAttachment, XmATTACH_FORM); ac++;
    XtSetArg(al[ac],XmNleftAttachment, XmATTACH_NONE); ac++;
    XtSetArg(al[ac],XmNbottomAttachment, XmATTACH_POSITION); ac++;
    XtSetArg(al[ac],XmNbottomPosition, 35); ac++;
    XtSetValues(t2range,al,ac);

    ac=0;
    XtSetArg(al[ac],XmNtopAttachment, XmATTACH_POSITION); ac++;
    XtSetArg(al[ac],XmNtopPosition, 40); ac++;
    XtSetArg(al[ac],XmNrightAttachment, XmATTACH_FORM); ac++;
    XtSetArg(al[ac],XmNleftAttachment, XmATTACH_FORM); ac++;
    XtSetArg(al[ac],XmNbottomAttachment, XmATTACH_NONE); ac++;
```

```
    XtSetValues(sep2,al,ac);

   ac=0;
   XtSetArg(al[ac],XmNtopAttachment, XmATTACH_WIDGET); ac++;
   XtSetArg(al[ac],XmNtopWidget, sep2); ac++;
   XtSetArg(al[ac],XmNleftAttachment, XmATTACH_FORM); ac++;
   XtSetArg(al[ac],XmNrightAttachment, XmATTACH_POSITION); ac++;
   XtSetArg(al[ac],XmNrightPosition, 40); ac++;
   XtSetArg(al[ac],XmNbottomAttachment, XmATTACH_POSITION); ac++;
   XtSetArg(al[ac],XmNbottomPosition, 45); ac++;
   XtSetValues(qrylabel,al,ac);

   ac=0;
   XtSetArg(al[ac],XmNtopAttachment, XmATTACH_WIDGET); ac++;
   XtSetArg(al[ac],XmNtopWidget, sep2); ac++;
   XtSetArg(al[ac],XmNleftAttachment, XmATTACH_POSITION); ac++;
   XtSetArg(al[ac],XmNleftPosition, 60); ac++;
   XtSetArg(al[ac],XmNrightAttachment, XmATTACH_FORM); ac++;
   XtSetArg(al[ac],XmNbottomAttachment, XmATTACH_POSITION); ac++;
   XtSetArg(al[ac],XmNbottomPosition, 45); ac++;
   XtSetValues(reslabel,al,ac);

   ac=0;
   XtSetArg(al[ac],XmNtopAttachment, XmATTACH_WIDGET); ac++;
   XtSetArg(al[ac],XmNtopWidget, sep2); ac++;
   XtSetArg(al[ac],XmNleftAttachment, XmATTACH_WIDGET); ac++;
   XtSetArg(al[ac],XmNleftWidget, qrylabel); ac++;
   XtSetArg(al[ac],XmNrightAttachment, XmATTACH_WIDGET); ac++;
   XtSetArg(al[ac],XmNrightWidget, reslabel); ac++;
   XtSetArg(al[ac],XmNbottomAttachment, XmATTACH_POSITION); ac++;
   XtSetArg(al[ac],XmNbottomPosition, 45); ac++;
   XtSetValues(jclabel,al,ac);

   ac=0;
   XtSetArg(al[ac],XmNtopAttachment, XmATTACH_WIDGET); ac++;
   XtSetArg(al[ac],XmNtopWidget, qrylabel); ac++;
   XtSetArg(al[ac],XmNleftAttachment, XmATTACH_FORM); ac++;
   XtSetArg(al[ac],XmNrightAttachment, XmATTACH_POSITION); ac++;
   XtSetArg(al[ac],XmNrightPosition, 40); ac++;
   XtSetArg(al[ac],XmNbottomAttachment, XmATTACH_FORM); ac++;
   XtSetValues(qry,al,ac);

   ac=0;
   XtSetArg(al[ac],XmNtopAttachment, XmATTACH_WIDGET); ac++;
   XtSetArg(al[ac],XmNtopWidget, reslabel); ac++;
   XtSetArg(al[ac],XmNleftAttachment, XmATTACH_POSITION); ac++;
   XtSetArg(al[ac],XmNleftPosition, 60); ac++;
   XtSetArg(al[ac],XmNrightAttachment, XmATTACH_FORM); ac++;
   XtSetArg(al[ac],XmNbottomAttachment, XmATTACH_FORM); ac++;
   XtSetValues(res,al,ac);

   ac=0;
   XtSetArg(al[ac],XmNtopAttachment, XmATTACH_WIDGET); ac++;
   XtSetArg(al[ac],XmNtopWidget, jclabel); ac++;
   XtSetArg(al[ac],XmNleftAttachment, XmATTACH_WIDGET); ac++;
   XtSetArg(al[ac],XmNleftWidget, qry); ac++;
```

```
    XtSetArg(al[ac],XmNrightAttachment, XmATTACH_WIDGET); ac++;
    XtSetArg(al[ac],XmNrightWidget, res); ac++;
    XtSetArg(al[ac],XmNbottomAttachment, XmATTACH_FORM); ac++;
    XtSetValues(jclst,al,ac);

    XtRealizeWidget(toplevel);
    XtAppMainLoop(context);
}
```

Appendix D C Code

Oct 11 17:44 1992 m_tree.c Page 1

```
/*****************************************************************************
                                    PROGRAM M_TREE.C
   Project specifications developed by:
         Peggy Wright in support of IAMS research
   Code written by:
         Shannon Thornton           19 AUG 92


   Contains the necessary routines to create, manipulate, and dispose of a
   binary tree.  Each node in the tree contains a column name, data type,
   data size, number of decimal places (if numeric), and a linked list of
   tables.  The list contains all table names in which the column is used.
   There is a linked list stored in each node of the table list.  This list
   contains the names of all indexes used in the current table that reference
   the current column.  See TYPES.H for the structure definitions.

                                Naming Convention
   Functions used for manipulating the tree or list structure begin with the
   first letter of the structure used.  For example, to insert a column into
   the column tree, c_ins(..) is called.  The c_ indicates the function uses
   the column tree structure.  All functions that manipulate the table list
   inside a node in the column tree begin with c_t.  Any function that
   manipulates the index list inside the previously mentioned table list
   begins with c_i.  All table tree functions start with t_, and all index
   tree functions start with i_.  This convention is also used for global
   tree variables, c_tree, t_tree, and i_tree, which reference the column
   tree, table tree, and index tree, respectively.
*****************************************************************************/

#include <string.h>
#include <stdio.h>
#include <malloc.h>
#include "mks.h"

/* - - - - - - - - - - - - - - * Global Tree Variables * - - - - - - - - - - - - - - */
struct m_node *m_tree;      /* Pointer to the column tree                          */
int cmp;                    /* Temporary variable used in find and insert          */
                            /* Since find and insert are recursive procedures      */
                            /* the temporary variable is declared global.  If      */
                            /* it was declared locally, it would be                */
                            /* re-allocated on the stack for every function        */
                            /* call.                                               */


/* - - - - - - - - - - - - - - * Function definitions * - - - - - - - - - - - - - -  */

/* - - - - - - - - - - - - - - - - * * * * * * * * * * - - - - - - - - - - - - - - - - */
/* - - - - - - - - - - - - - - - - * Tree  Functions * - - - - - - - - - - - - - - - - */
/* - - - - - - - - - - - - - - - - * * * * * * * * * * - - - - - - - - - - - - - - - - */
m_shownode(info)
struct mks info;
/*****************************************************************************
DESCRIPTION:      Prints the contents of info to standard
                  output.
PARAMETERS:       struct column info
RETURNS:          None
```

```
CALLS:              None
CALLED BY:          c_lnr
*****************************************************************************/
{
   printf("%-30s %-9s %3d ",info.name, info.type, info.size);
   if (info.type[0] == 'N') printf("%d\n",info.decimal);
   else printf("\n");
}

m_init()
/*****************************************************************************
DESCRIPTION:        Initializes the global tree variable c_tree
                    to NULL.
PARAMETERS:         None
RETURNS:            None
CALLS:              None
CALLED BY:          (KS.PC) init_trees
*****************************************************************************/
{
   m_tree = NULL;
}

m_store(t, info)
struct m_node *t;
struct mks info;
/*****************************************************************************
DESCRIPTION:        Puts the contents of info into the info
                    field of the c_node pointer t.
PARAMETERS:         struct c_node *t, struct column info
RETURNS:            None
CALLS:              None
CALLED BY:          c_ins
*****************************************************************************/
{
   t->parent = NULL;
   t->lchild = NULL;
   t->rchild = NULL;
   t->info = info;
}

struct m_node *m_ins(t, info)
struct m_node **t;
struct mks info;
/*****************************************************************************
DESCRIPTION:        Creates a node in the tree pointed to by t,
                    and stores the contents of info in that
                    node.  If a node already exists with the
                    same key field, a message is printed
                    warning the user.  If there is insufficient
                    memory to allocate the new node, a warning
                    is printed.
PARAMETERS:         struct c_node **t, struct column info
RETURNS:            Pointer to the new node.  If a node already
                    exists with the same key field, a pointer
                    to that node is returned.  If there is
                    insufficient memory, NULL is returned.
```

Appendix D C Code

Oct 11 17:44 1992 m_tree.c Page 3

```
CALLS:              c_store, c_tinit, c_ins
CALLED BY:          (KS.PC) get_column_info
*****************************************************************************/
{
   if (*t == NULL) {
      *t = (struct m_node *) malloc(sizeof(struct m_node));
      if (*t == NULL) {
         printf("(MKS insert):  Out of memory\n");
         return NULL;
      }
      m_store(*t, info);
      m_ainit(&(*t)->info);
      return *t;
   }
   else {
      cmp = strcmp(info.name, (*t)->info.name);
      if (cmp < 0) return m_ins(&(*t)->lchild, info);
      else
         if (cmp > 0) return m_ins(&(*t)->rchild, info);
         else {
            printf("*** DUPLICATE KEY (%s) ***\n",info.name);
            return *t;
         }
   }
}

m_deltree(t)
struct m_node *t;
/*****************************************************************************
DESCRIPTION:        Deletes the entire tree pointed to by t.
PARAMETERS:         struct c_node *t
RETURNS:            None
CALLS:              c_deltree, c_tdeltree
CALLED BY:          (KS.PC) del_trees
*****************************************************************************/
{
   if (t->lchild != NULL) {
      m_deltree(t->lchild);
      t->lchild = NULL;
   }
   if (t->rchild != NULL) {
      m_deltree(t->rchild);
      t->rchild = NULL;
   }
   m_adeltest(&t->info, t->info.app_head);
   free(t);
}

struct m_node *m_find(t, name)
struct m_node *t;
char *name;
/*****************************************************************************
DESCRIPTION:        Searches the tree pointed to by t for the
                    node containing name.
PARAMETERS:         struct c_node *t, char *name
RETURNS:            Pointer to the node containing name, or
```

```
                    NULL if not found
CALLS:              c_find
CALLED BY:          c_find
*****************************************************************************/
{
   if (t == NULL) return NULL;
   cmp = strcmp(name,t->info.name);
   if (cmp < 0) return m_find(t->lchild, name);
   else
      if (cmp > 0) return m_find(t->rchild, name);
      else return t;
}

struct m_node *m_left(t)
struct m_node **t;
/*****************************************************************************
DESCRIPTION:        Changes t to point to its own left child.
                    If t is NULL upon calling the function,
                    nothing is changed.
PARAMETERS:         struct c_node **t
RETURNS:            Returns a pointer to t's left child.  If t
                    is NULL, NULL is returned.
CALLS:              None
CALLED BY:          None
*****************************************************************************/
{
   if (*t != NULL) *t = (*t)->lchild;
   return *t;
}

struct m_node *m_right(t)
struct m_node **t;
/*****************************************************************************
DESCRIPTION:        Changes t to point to its own right child.
                    If t is NULL upon calling the function,
                    nothing is changed.
PARAMETERS:         struct node **t
RETURNS:            Returns a pointer to t's right child.  If t
                    is NULL, NULL is returned.
CALLS:              None
CALLED BY:          None
*****************************************************************************/
{
   if (*t != NULL) *t = (*t)->rchild;
   return *t;
}

m_lnr(t)
struct m_node *t;
/*****************************************************************************
DESCRIPTION:        Performs an Inorder traversal (LNR) of the
                    tree pointed to by t while displaying the
                    contents of each node as it is visited.
PARAMETERS:         struct c_node *t
RETURNS:            None
CALLS:              c_lnr, c_shownode
```

```
CALLED BY:        c_lnr
*****************************************************************************/
{
   if (t->lchild != NULL) m_lnr(t->lchild);
   m_shownode(t->info);
   if (t->rchild != NULL) m_lnr(t->rchild);
}

/* - - - - - - - - - - - - - -* * * * * * * * * *- - - - - - - - - - - - - - */
/* - - - - - - - - - - - - - -* Application List *- - - - - - - - - - - - - - */
/* - - - - - - - - - - - - - -*     Functions     *- - - - - - - - - - - - - - */
/* - - - - - - - - - - - - - -* * * * * * * * * *- - - - - - - - - - - - - - */
m_ainit(m)
struct mks *m;
/*****************************************************************************
DESCRIPTION:      Initializes the table list pointers:
                  tbl_cur, tbl_head, and tbl_tail.
PARAMETERS:       struct column *c
RETURNS:          None
CALLS:            None
CALLED BY:        c_ins
*****************************************************************************/
{
   m->app_cur = m->app_head = m->app_tail = NULL;
}

m_ains(m, info)
struct mks *m;
struct m_app_info info;
/*****************************************************************************
DESCRIPTION:      Creates a new node at the end of the table
                  list and stores the contents of info in the
                  node.  If there is insufficient memory to
                  create the node, a message is printed.
PARAMETERS:       struct column *c, struct c_tab_info info
RETURNS:          None
CALLS:            c_iinit
CALLED BY:        (KS.PC) get_column_info
*****************************************************************************/
{
   if (m->app_tail != NULL) {
      m->app_tail->next = (struct m_anode *) malloc(sizeof(struct m_anode));
      if (m->app_tail->next == NULL) {
         printf("(MKS/application list insert):  Out of memory\n");
         return NULL;
      }
      m->app_tail->next->prev = m->app_tail;
      m->app_tail = m->app_tail->next;
      m->app_tail->info = info;
      m->app_tail->next = NULL;
   }
   else {
      m->app_tail = (struct m_anode *) malloc(sizeof(struct m_anode));
      if (m->app_tail == NULL) {
         printf("(MKS/application list insert):  Out of memory\n");
         return NULL;
```

```
        }
        m->app_head = m->app_tail;
        m->app_tail->next = NULL;
        m->app_head->prev = NULL;
        m->app_tail->info = info;
    }
}

m_adel(m, node)
struct mks *m;
struct m_anode *node;
/*****************************************************************************
DESCRIPTION:        Deletes node, pointed to by node, from the
                    table list.
PARAMETERS:         struct column *c, struct c_tnode *node
RETURNS:            None
CALLS:              c_idelrest
CALLED BY:          c_tdelrest
*****************************************************************************/
{
    if (node == NULL) return;
    if ((node == m->app_head) || (node == m->app_tail)) {
        if (node == m->app_head) {
            m->app_head = node->next;
            if (m->app_head != NULL) m->app_head->prev = NULL;
        }
        if (node == m->app_tail) {
            m->app_tail = node->prev;
            if (m->app_tail != NULL) m->app_tail->next = NULL;
        }
    }
    else {
        node->prev->next = node->next;
        node->next->prev = node->prev;
    }
    free(node);
}

m_adelrest(m, node)
struct mks *m;
struct m_anode *node;
/*****************************************************************************
DESCRIPTION:        Deletes all nodes from node to tbl_tail
                    (current node to end of list) from the
                    table list.
PARAMETERS:         struct column *c, struct c_tnode *node
RETURNS:            None
CALLS:              c_tdel
CALLED BY:          c_deltree
*****************************************************************************/
{
    struct m_anode *tmp;

    while (node != NULL) {
        tmp = node->next;
        m_adel(m, node);
```

```
        node = tmp;
    }
}

struct m_anode *m_afind(m, name)
struct mks *m;
char *name;
/*****************************************************************************
DESCRIPTION:        Searches the table list for a node
                    containing name.
PARAMETERS:         struct column *c, char *name
RETURNS:            Pointer to the next node
CALLS:              None
CALLED BY:          None
*****************************************************************************/
{
    struct m_anode *tmp;
}

struct m_anode *m_anext(m)
struct mks *m;
/*****************************************************************************
DESCRIPTION:        Advances the tbl_cur pointer in c to the
                    next node in the table list.
PARAMETERS:         struct column *c
RETURNS:            Pointer to the next node
CALLS:              None
CALLED BY:          show_column_info
*****************************************************************************/
{
    if (m->app_cur != NULL) m->app_cur = m->app_cur->next;
}

struct m_anode *m_aprev(m)
struct mks *m;
/*****************************************************************************
DESCRIPTION:        Move the tbl_cur pointer in c to the
                    previous node in the table list.
PARAMETERS:         struct column *c
RETURNS:            Pointer to the previous node
CALLS:              None
CALLED BY:          None
*****************************************************************************/
{
    if (m->app_cur != NULL) m->app_cur = m->app_cur->prev;
}
```

Appendix D C Code

Oct 11 17:48 1992 mks.h Page 1

```
/*****************************************************************************
                                    FILE MKS.H
   IAMS meta knowledge source header file.
   Project specifications developed by:
         Peggy Wright in support of IAMS research
   Code written by:
         Shannon Thornton           19 AUG 92


   Contains all data types used to model IAMS Knowledge Source.  The file
   consists of three parts:  column structures, table structure and index
   structures.  At the beginning of each part there is a BNF-like description
   of the particular structure.

                                Naming Convention
   Functions used for manipulating the tree or list structure begin with the
   first letter of the structure used.  For example, to insert a column into
   the column tree, c_ins(..) is called.  The c_ indicates the function uses
   the column tree structure.  All functions that manipulate the table list
   inside a node in the column tree begin with c_t.  Any function that
   manipulates the index list inside the previously mentioned table list
   begins with c_i.  All table tree functions start with t_, and all index
   tree functions start with i_.  This convention is also used for global
   tree variables, c_tree, t_tree, and i_tree, which reference the column
   tree, table tree, and index tree, respectively.
*****************************************************************************/

#include "iams_const.h"            /* Include IAMS Constants file            */

/*****************************************************************************/
/* - - - - - - - - - - - - - - -* * * * * * * * *- - - - - - - - - - - - - - */
/* - - - - - - - - - - - - - - -* MKS  Structures *- - - - - - - - - - - - - */
/* - - - - - - - - - - - - - - -* * * * * * * * *- - - - - - - - - - - - - - */
/*****************************************************************************
                  BNF-like Description of the Column Structures

  <Column>          ::=  <column name> <type> <size> [decimal]
                         { <app. name> }
  <column name>     ::=  Any valid Oracle column name
  <app. name>       ::=  Any valid table space name
  <type>            ::=   Char   | Date | Numeric |    Long
  <size>            ::=  1..240 |    7   |   1..38   | 0..65536
  [decimal]         ::=                  | -37..37 |
  *** NOTE ***            Raw and Long Raw types are also acceptible.  The
                          Raw type is similar to Char, and the Long Raw is
                          similar to Long.  The same size restrictions apply.
                          See Oracle Database Administrator's Guide for more
                          information.

  Conventions used in BNF-like syntax:
     { } - indicates the element(s) can occur zero or more times.
     < > - indicates the element is a non-terminal
     [ ] - indicates the element is a non-terminal that is optional in the
           definition
      |  - Synonym for the OR operator.  X | Y is read "X or Y"
     ' ' - Indicates character data is stored
```

Oct 11 17:48 1992 mks.h Page 2

```
*****************************************************************************/

struct m_app_info {
   char name[NAMELENGTH];          /* Application name                       */
};

struct m_anode {                   /* Linked list node used in mks to store  */
                                   /* the applications in which the current  */
                                   /* column is used.                        */
   struct m_app_info info;         /* Data stored in the node                */
   struct m_anode *prev;           /* Pointer to the previous node           */
   struct m_anode *next;           /* Pointer to the next node               */
};

struct mks {
   char name[NAMELENGTH];          /* Column name                            */
   char type[10];                  /* Column type                            */
   int size;                       /* Column size                            */
   int decimal;                    /* Number of decimal places if column is  */
                                   /* numeric                                */
   struct m_anode *app_head;       /* Pointer to a linked list of apps in    */
                                   /* which the column appears               */
   struct m_anode *app_tail;       /* Pointer to the tail of the linked list */
                                   /* of applications.                       */
   struct m_anode *app_cur;        /* Pointer to current node in the linked  */
                                   /* list.  Used for traversal.             */
};

struct m_node {                    /* Tree node used to store information    */
                                   /* about each column                      */
   struct m_node *parent;          /* Pointer to the parent node             */
   struct mks info;                /* Data stored in the node                */
   struct m_node *lchild;          /* Pointer to the left sub-tree           */
   struct m_node *rchild;          /* Pointer to the right sub-tree          */
};

/* -------------------------* End of MKS Structures *----------------------- */
```

Appendix D C Code

Oct 11 17:47 1992 qc.c Page 1

```
/*************************************************************************
                                    PROGRAM QC.C

   IAMS query constructor builds SQL queries.
   Project specifications developed by:
         Peggy Wright in support of IAMS research
   Code written by:
         Shannon Thornton            03 AUG 92

    INPUT:   table names
             column names & column values
*************************************************************************/
#include <stdio.h>
#include "types.h"

extern struct c_node *c_tree;

extern FILE *iams_qry, *iams_log;

char *check_type(c, r)
struct c_node *c;
char *r;
{
   static char temp[50];

   switch (c->info.type[0]) {
      case 'N':
         strcpy(temp,r);
         break;
      case 'D':
         sprintf(temp,"TO_DATE(%s, 'MM/DD/YY')",r);
         break;
      default:
         sprintf(temp,"'%s'",r);
   }
   return temp;
}

bld_query(query, type, t1, t1op, t1range, t1table, t2, t2op, t2range,
   t2table, jc, jcop, jcrange)
char *query, *t1, *t1op, *t1range, *t1table, *t2, *t2op, *t2range, *t2table,
   *jc, *jcop, *jcrange;
int type;
/*************************************************************************
DESCRIPTION:
PARAMETERS:
RETURNS:
CALLS:
CALLED BY:
*************************************************************************/
{
   struct c_node *t1node, *t2node, *jcnode;

   if (t1[0] != NULL)
      t1node = (struct c_node *) c_find(c_tree, t1);
   if (t2[0] != NULL)
```

```
      t2node = (struct c_node *) c_find(c_tree, t2);
   if (jc[0] != NULL)
      jcnode = (struct c_node *) c_find(c_tree, jc);
   switch (type) {
      case TBL:
         if ((t1[0] != NULL) && (t2[0] != NULL))
            sprintf(query,"SELECT %s, %s FROM %s",t1,t2,t1table);
         else if (t1[0] != NULL)
            sprintf(query,"SELECT %s FROM %s", t1, t1table);
         else sprintf(query,"SELECT %s FROM %s", t2, t2table);
         if (t1node != NULL) {
            if (t1op[0] != NULL) {
               sprintf(query,"%s WHERE %s %s %s",query,t1,t1op,
                  check_type(t1node,t1range));
               if (t2node != NULL)
                  if (t2op[0] != NULL)
                     sprintf(query,"%s AND %s %s %s",query,t2,t2op,
                        check_type(t2node,t2range));
            }
            else if (t2node != NULL)
               if (t2op[0] != NULL)
                  sprintf(query,"%s WHERE %s %s %s",query,t2,t2op,
                     check_type(t2node,t2range));
         }
         else if (t2node != NULL)
            sprintf(query,"%s WHERE %s %s %s",query,t2,t2op,
               check_type(t2node,t2range));
         break;
      case TBL_COL_TBL:
         sprintf(query,"SELECT %s.%s, %s.%s, %s.%s FROM %s, %s", t1table, t1,
            t2table, jc, t2table, t2, t1table, t2table);
         if (t1op[0] != NULL) {
            sprintf(query, "%s WHERE %s.%s %s %s", query, t1table, t1,
               t1op, check_type(t1node,t1range));
            if (t2op[0] != NULL)
               sprintf(query,"%s AND %s.%s %s %s", query, t2table, t2,
                  t2op, check_type(t2node,t2range));
         }
         else if (t2op[0] != NULL)
            sprintf(query,"%s WHERE %s.%s %s %s", query, t2table, t2,
               t2op, check_type(t2node,t2range));
         if (jcop[0] != NULL)
            if ((t1op[0] != NULL) || (t2op[0] != NULL))
               sprintf(query,"%s AND %s.%s %s %s AND %s.%s = %s.%s",
                  query, t1table, jc, jcop, check_type(jcnode,jcrange),
                  t1table, jc, t2table, jc);
            else
               sprintf(query,"%s WHERE %s.%s %s %s AND %s.%s = %s.%s",
                  query, t1table, jc, jcop, check_type(jcnode,jcrange),
                  t1table, jc, t2table, jc);
         else
            if ((t1op[0] != NULL) || (t2op[0] != NULL))
               sprintf(query,"%s AND %s.%s = %s.%s", query, t1table, jc,
                  t2table, jc);
            else
               sprintf(query,"%s WHERE %s.%s = %s.%s", query, t1table, jc,
```

```
                    t2table, jc);
         break;
      default:
         query = NULL;
   }
   trim(query);
   fprintf(iams_qry, "%s\n", query);
   fprintf(iams_log, "%s\n", query);
}
```

Appendix D C Code

Oct 11 17:44 1992 strutil.c Page 1

```
/*****************************************************************************
                                    PROGRAM STRUTIL.C

   Project specifications developed by:
         Peggy Wright in support of IAMS research
   Code written by:
         Shannon Thornton           03 AUG 92
   Shannon Thornton

   Contains the following helpful character string functions:

   rtrim  - Removes trailing spaces from a character string
   ltrim  - Removes leading spaces from a character string
   trim   - Removes leading and trailing spaces from a character string
   strupr - Convert a character string to upper case
   strlwr - Convert a character string to lower case

*****************************************************************************/
#include <string.h>
#include <ctype.h>

char *rtrim(s)
char *s;
/*****************************************************************************
DESCRIPTION:      Removes trailing spaces from the character
                  string s.
PARAMETERS:       char *s
RETURNS:          Pointer to the string with the spaces
                  removed.
CALLS:            None
CALLED BY:        None
*****************************************************************************/
{
   int i = (strlen(s) - 1);

   if (s == NULL) return s;
   while ((s[i] == ' ') && (i >= 0)) s[i--] = '\0';
   return s;
}

char *ltrim(s)
char *s;
/*****************************************************************************
DESCRIPTION:      Removes preceding spaces from a character
                  string s.
PARAMETERS:       char *s
RETURNS:          Pointer to a string with the spaces
                  removed.
CALLS:            None
CALLED BY:        None
*****************************************************************************/
{
   int i = 0;

   if (s == NULL) return s;
   while ((s[i] == ' ') && (i <= strlen(s))) i++;
   if (i > 0) strcpy(s,(char *) &s[i]);
   return s;
}
```

```
char *trim(s)
char *s;
/*****************************************************************************
DESCRIPTION:      Removes preceding and trailing spaces from
                  a character string s.
PARAMETERS:       char *s
RETURNS:          Pointer to a string with the spaces
                  removed.
CALLS:            ltrim, rtrim
CALLED BY:        None
*****************************************************************************/
{
   if (s == NULL) return s;
   ltrim(s);
   rtrim(s);
   return s;
}

char *strupr(s)
char *s;
/*****************************************************************************
DESCRIPTION:      Converts a character string s to all upper
                  case.
PARAMETERS:       char *s
RETURNS:          Pointer to a string converted to upper
                  case.
CALLS:            None
CALLED BY:        None
*****************************************************************************/
{
   int i;

   if (s == NULL) return s;
   for (i=0;i<strlen(s);i++) s[i] = toupper(s[i]);
   return s;
}

char *strlwr(s)
char *s;
/*****************************************************************************
DESCRIPTION:      Converts a character string s to all lower
                  case.
PARAMETERS:       char *s
RETURNS:          Pointer to a string converted to lower
                  case.
CALLS:            None
CALLED BY:        None
*****************************************************************************/
{
   int i;

   if (s == NULL) return s;
   for (i=0;i<strlen(s);i++) s[i] = tolower(s[i]);
   return s;
}
```

Appendix D C Code

Oct 11 17:47 1992 t_tree.c Page 1

```
/*****************************************************************************
                                    PROGRAM T_TREE.C
   Project specifications developed by:
         Peggy Wright in support of IAMS research
   Code written by:
         Shannon Thornton           03 AUG 92

   Contains the necessary routines to create, manipulate, and dispose of a
   binary tree.  Each node in the tree contains a table name, a linked list
   of columns, and a linked list of indexes.  The column list contains the
   column names of all columns used in the table, while the index list
   contains the index names of all indexes used in the table.  See TYPES.H
   for structure definitions.

                                Naming Convention
   Functions used for manipulating the tree or list structure begin with the
   first letter of the structure used.  For example, to insert a column into
   the column tree, c_ins(..) is called.  The c_ indicates the function uses
   the column tree structure.  All functions that manipulate the table list
   inside a node in the column tree begin with c_t.  Any function that
   manipulates the index list inside the previously mentioned table list
   begins with c_i.  All table tree functions start with t_, and all index
   tree functions start with i_.  This convention is also used for global
   tree variables, c_tree, t_tree, and i_tree, which reference the column
   tree, table tree, and index tree, respectively.
*****************************************************************************/

#include <string.h>
#include <stdio.h>
#include <malloc.h>
#include "types.h"


/* - - - - - - - - - - - - - - * Global Tree Variables * - - - - - - - - - - - - - */
struct t_node *t_tree;      /* Pointer to the table tree                        */
int cmp;                    /* Temporary variable used in find and insert       */
                            /* Since find and insert are recursive procedures */
                            /* the temporary variable is declared global.  If */
                            /* it was declared locally, it would be           */
                            /* re-allocated on the stack for every function   */
                            /* call.                                          */


/* - - - - - - - - - - - - - - * Function definitions * - - - - - - - - - - - - -  */

/* - - - - - - - - - - - - - - - - * * * * * * * * * * - - - - - - - - - - - - - - - */
/* - - - - - - - - - - - - - - - - * Tree  Functions * - - - - - - - - - - - - - - - */
/* - - - - - - - - - - - - - - - - * * * * * * * * * * - - - - - - - - - - - - - - - */
t_shownode(info)
struct table info;
/*****************************************************************************
DESCRIPTION:      Prints the contents of info to standard
                  output.
PARAMETERS:       struct table info
RETURNS:          None
CALLS:            None
```

```
CALLED BY:        t_lnr
*****************************************************************************/
{
   printf("%-32s\n",info.name);
}

 t_init()
/*****************************************************************************
DESCRIPTION:      Initializes the global tree variable c_tree
                  to NULL.
PARAMETERS:       None
RETURNS:          None
CALLS:            None
CALLED BY:        (KS.PC) init_trees
*****************************************************************************/
{
   t_tree = NULL;
}

t_store(t, info)
struct t_node *t;
struct table info;
/*****************************************************************************
DESCRIPTION:      Puts the contents of info into the info
                  field of the t_node pointer t.
PARAMETERS:       struct t_node *t, struct table info
RETURNS:          None
CALLS:            None
CALLED BY:        t_ins
*****************************************************************************/
{
   t->parent = NULL;
   t->lchild = NULL;
   t->rchild = NULL;
   t->info = info;
}

struct t_node *t_ins(t, info)
struct t_node **t;
struct table info;
/*****************************************************************************
DESCRIPTION:      Creates a node in the tree pointed to by t,
                  and stores the contents of info in that
                  node.  If a node already exists with the
                  same key field, a message is printed
                  warning the user.  If there is insufficient
                  memory to allocate the new node, a warning
                  is printed.
PARAMETERS:       struct t_node **t, struct table info
RETURNS:          Pointer to a new node containing info.  If
                  a node already exists with the same key
                  field, a pointer to that node is returned.
                  If there is insufficient memory, NULL is
                  returned.
CALLS:            t_ins, t_store, t_cinit, t_init
CALLED BY:        (KS.PC) get_table_info, t_ins
```

Oct 11 17:47 1992 t_tree c Page 3

```
*****************************************************************************/
{
   if (*t == NULL) {
      *t = (struct t_node *) malloc(sizeof(struct t_node));
      if (*t == NULL) {
         printf("(table insert): Out of memory\n");
         return NULL;
      }
      t_store(*t, info);
      t_cinit(&(*t)->info);
      t_iinit(&(*t)->info);
      return *t;
   }
   else {
      cmp = strcmp(info.name, (*t)->info.name);
      if (cmp < 0) return t_ins(&(*t)->lchild, info);
      else
         if (cmp > 0) return t_ins(&(*t)->rchild, info);
         else {
            printf("*** DUPLICATE KEY (%s) ***\n",info.name);
            return *t;
         }
   }
}

t_deltree(t)
struct t_node *t;
/*****************************************************************************
DESCRIPTION:      Deletes the tree pointed to by t.
PARAMETERS:       struct t_node *t
RETURNS:          None
CALLS:            t_deltree, t_cdelrest, t_idelrest
CALLED BY:        (KS.PC, del_trees
*****************************************************************************/
{
   if (t->lchild != NULL) {
      t_deltree(t->lchild);
      t->lchild = NULL;
   }
   if (t->rchild != NULL) {
      t_deltree(t->rchild);
      t->rchild = NULL;
   }
   t_cdelrest(&t->info, t->info.col_head);
   t_idelrest(&t->info, t->info.idx_head);
   free(t);
}

struct t_node *t_find(t, name)
struct t_node *t;
char *name;
/*****************************************************************************
DESCRIPTION:      Searches the tree pointed to by t for the
                  node containing name.
PARAMETERS:       struct t_node *t, char *name
RETURNS:          Pointer to the node containing name, or
```

```
                    NULL if not found
CALLS:              t_find
CALLED BY:          t_find
*****************************************************************************/
{
   if (t == NULL) return NULL;
   cmp = strcmp(name,t->info.name);
   if (cmp < 0) return t_find(t->lchild, name);
   else
      if (cmp > 0) return t_find(t->rchild, name);
      else return t;
}

struct t_node *t_left(t)
struct t_node **t;
/*****************************************************************************
DESCRIPTION:        Changes t to point to its own left child.
                    If t is NULL upon calling the function,
                    nothing is changed.
PARAMETERS:         struct t_node **t
RETURNS:            Returns a pointer to t's left child.  If t
                    is NULL, NULL is returned.
CALLS:              None
CALLED BY:          None
*****************************************************************************/
{
   if (*t != NULL) *t = (*t)->lchild;
   return *t;
}

struct t_node *t_right(t)
struct t_node **t;
/*****************************************************************************
DESCRIPTION:        Changes t to point to its own right child.
                    If t is NULL upon calling the function,
                    nothing is changed.
PARAMETERS:         struct t_node **t
RETURNS:            Returns a pointer to t's right child.  If t
                    is NULL, NULL is returned.
CALLS:              None
CALLED BY:          None
*****************************************************************************/
{
   if (*t != NULL) *t = (*t)->rchild;
   return *t;
}

t_lnr(t)
struct t_node *t;
/*****************************************************************************
DESCRIPTION:        Performs an Inorder traversal (LNR) of the
                    tree pointed to by t while displaying the
                    contents of each node as it is visited.
PARAMETERS:         struct t_node *t
RETURNS:            None
CALLS:              t_lnr, t_shownode
```

```
CALLED BY:        t_lnr
*****************************************************************************/
{
   if (t->lchild != NULL) t_lnr(t->lchild);
   t_shownode(t->info);
   if (t->rchild != NULL) t_lnr(t->rchild);
}

/* - - - - - - - - - - - - - - * * * * * * * * * * - - - - - - - - - - - - - - */
/* - - - - - - - - - - - - - -  *   Table List    * - - - - - - - - - - - - - */
/* - - - - - - - - - - - - - -  *    Functions    * - - - - - - - - - - - - - */
/* - - - - - - - - - - - - - - * * * * * * * * * * - - - - - - - - - - - - - - */
t_cinit(t)
struct table *t;
/*****************************************************************************
DESCRIPTION:      Initializes the column list pointers:
                  col_cur, col_head, and col_tail.
PARAMETERS:       struct table *t
RETURNS:          None
CALLS:            None
CALLED BY:        t_ins
*****************************************************************************/
{
   t->col_cur = t->col_head = t->col_tail = NULL;
}

t_cins(t, info)
struct table *t;
struct t_col_info info;
/*****************************************************************************
DESCRIPTION:      Creates a new node at the end of the column
                  list and stores the contents of info in the
                  node.  If there is insufficient memory to
                  create the node, a message is printed.
PARAMETERS:       struct table *t, struct t_col_info info
RETURNS:          None
CALLS:            None
CALLED BY:        (KS.PC) get_table_info
*****************************************************************************/
{
   if (t->col_tail != NULL) {
      t->col_tail->next = (struct t_cnode *) malloc(sizeof(struct t_cnode));
      if (t->col_tail->next == NULL) {
         printf("(table/column list insert):  Out of memory\n");
         return NULL;
      }
      t->col_tail->next->prev = t->col_tail;
      t->col_tail = t->col_tail->next;
      t->col_tail->info = info;
      t->col_tail->next = NULL;
   }
   else {
      t->col_tail = (struct t_cnode *) malloc(sizeof(struct t_cnode));
      if (t->col_tail == NULL) {
         printf("(table/column list insert):  Out of memory\n");
         return NULL;
```

```
        }
        t->col_head = t->col_tail;
        t->col_tail->next = NULL;
        t->col_head->prev = NULL;
        t->col_tail->info = info;
    }
}

t_cdel(t, node)
struct table *t;
struct t_cnode *node;
/*****************************************************************************
DESCRIPTION:        Deletes node from the column list.
PARAMETERS:         struct table *t, struct t_cnode *node
RETURNS:            None
CALLS:              None
CALLED BY:          t_cdelrest
*****************************************************************************/
{
    if (node == NULL) return;
    if ((node == t->col_head) || (node == t->col_tail)) {
        if (node == t->col_head) {
            t->col_head = node->next;
            if (t->col_head != NULL) t->col_head->prev = NULL;
        }
        if (node == t->col_tail) {
            t->col_tail = node->prev;
            if (t->col_tail != NULL) t->col_tail->next = NULL;
        }
    }
    else {
        node->prev->next = node->next;
        node->next->prev = node->prev;
    }
    free(node);
}

t_cdelrest(t, node)
struct table *t;
struct t_cnode *node;
/*****************************************************************************
DESCRIPTION:        Deletes all nodes from node to col_tail
                    (current node to end of list) from the
                    column list.
PARAMETERS:         struct table *t, struct t_cnode *node
RETURNS:            None
CALLS:              t_cdel
CALLED BY:          t_deltree
*****************************************************************************/
{
    struct t_cnode *tmp;

    while (node != NULL) {
        tmp = node->next;
        t_cdel(t, node);
        node = tmp;
```

```
    }
}

struct t_cnode *t_cfind(t, name)
struct table *t;
char name[31];
/*****************************************************************************
DESCRIPTION:      Searches the column list for the node
                  containing name.
PARAMETERS:       struct table *t, char *name
RETURNS:          Pointer to a node containing name
CALLS:            None
CALLED BY:        None
*****************************************************************************/
{
   struct t_cnode *tmp;
}

struct t_cnode *t_cnext(t)
struct table *t;
/*****************************************************************************
DESCRIPTION:      Advances the col_cur pointer in t to the
                  next node in the column list.
PARAMETERS:       struct table *t
RETURNS:          Pointer to the next node
CALLS:            None
CALLED BY:        (KS.PC) show_table_info
*****************************************************************************/
{
   if (t->col_cur != NULL) t->col_cur = t->col_cur->next;
}

struct t_cnode *t_cprev(t)
struct table *t;
/*****************************************************************************
DESCRIPTION:      Moves the col_cur pointer in t to the
                  previous node in the column list.
PARAMETERS:       struct table *t
RETURNS:          Pointer to the previous node
CALLS:            None
CALLED BY:        None
*****************************************************************************/
{
   if (t->col_cur != NULL) t->col_cur = t->col_cur->prev;
}

/* - - - - - - - - - - - - - - * * * * * * * * * * - - - - - - - - - - - - - - */
/* - - - - - - - - - - - - - -  *   Index List    * - - - - - - - - - - - - - */
/* - - - - - - - - - - - - - -  *    Functions    * - - - - - - - - - - - - - */
/* - - - - - - - - - - - - - - * * * * * * * * * * - - - - - - - - - - - - - - */
t_iinit(t)
struct table *t;
/*****************************************************************************
DESCRIPTION:      Initializes the index list pointers stored
                  in t.
PARAMETERS:       struct table *c
```

```
RETURNS:            None
CALLS:              None
CALLED BY:          t_ins
*****************************************************************************/
{
   t->idx_cur = t->idx_head = t->idx_tail = NULL;
}

t_iins(t, info)
struct table *t;
struct t_idx_info info;
/*****************************************************************************
DESCRIPTION:        Creates a new node in the index list and
                    stores the contents of info in that node.
                    If there is insufficient memory to create
                    the new node, a message is printed.
PARAMETERS:         struct table *t, struct t_idx_info info
RETURNS:            None
CALLS:              None
CALLED BY:          (KS.PC) get_table_info
*****************************************************************************/
{
   if (t->idx_tail != NULL) {
      t->idx_tail->next = (struct t_inode *) malloc(sizeof(struct t_inode));
      if (t->idx_tail->next == NULL) {
         printf("(table/index list insert):  Out of memory\n");
         return NULL;
      }
      t->idx_tail->next->prev = t->idx_tail;
      t->idx_tail = t->idx_tail->next;
      t->idx_tail->info = info;
      t->idx_tail->next = NULL;
   }
   else {
      t->idx_tail = (struct t_inode *) malloc(sizeof(struct t_inode));
      if (t->idx_tail == NULL) {
         printf("(table/index insert):  Out of memory\n");
         return NULL;
      }
      t->idx_head = t->idx_tail;
      t->idx_tail->next = NULL;
      t->idx_head->prev = NULL;
      t->idx_tail->info = info;
   }
}

t_idel(t, node)
struct table *t;
struct t_inode *node;
/*****************************************************************************
DESCRIPTION:        Deletes node from the index list.
PARAMETERS:         struct table *t, struct t_inode *node
RETURNS:            None
CALLS:              None
CALLED BY:          t_idelrest
*****************************************************************************/
```

```
{
   if (node == NULL) return;
   if ((node == t->idx_head) || (node == t->idx_tail)) {
      if (node == t->idx_head) {
         t->idx_head = node->next;
         if (t->idx_head != NULL) t->idx_head->prev = NULL;
      }
      if (node == t->idx_tail) {
         t->idx_tail = node->prev;
         if (t->idx_tail != NULL) t->idx_tail->next = NULL;
      }
   }
   else {
      node->prev->next = node->next;
      node->next->prev = node->prev;
   }
   free(node);
}

t_idelrest(t, node)
struct table *t;
struct t_inode *node;
/*****************************************************************************
DESCRIPTION:      Deletes all nodes from the index list that
                  come after and including node.
PARAMETERS:       struct table *t, struct t_inode *node
RETURNS:          None
CALLS:            t_idel
CALLED BY:        t_deltree
*****************************************************************************/
{
   struct t_inode *tmp;

   while (node != NULL) {
      tmp = node->next;
      t_idel(t, node);
      node = tmp;
   }
}

struct t_inode *t_ifind(t, name)
struct table *t;
char name[31];
/*****************************************************************************
DESCRIPTION:      Searches the index list for the node
                  containing name.
PARAMETERS:       struct table *t, char *name
RETURNS:          Pointer to a node containing name
CALLS:            None
CALLED BY:        None
*****************************************************************************/
{
   struct t_inode *tmp;
}

struct t_inode *t_inext(t)
```

```
struct table *t;
/*****************************************************************************
DESCRIPTION:        Advances the idx_cur pointer stored in t to
                    the next node in the index list.
PARAMETERS:         struct table *t
RETURNS:            Pointer to the next node in the index list.
CALLS:              None
CALLED BY:          (KS.PC) show_table_info
*****************************************************************************/
{
   if (t->idx_cur != NULL) t->idx_cur = t->idx_cur->next;
}

struct t_inode *t_iprev(t)
struct table *t;
/*****************************************************************************
DESCRIPTION:        Changes the idx_cur pointer stored in t to
                    the previous node in the index list.
PARAMETERS:         struct table *t
RETURNS:            Pointer to the previous node in the index
                    list.
CALLS:              None
CALLED BY:          None
*****************************************************************************/
{
   if (t->idx_cur != NULL) t->idx_cur = t->idx_cur->prev;
}
```

Appendix D C Code

Oct 11 17:46 1992 types.h Page 1

```
/*****************************************************************************
                                    File types.h
   Specifications developed by:
         Peggy Wright in support of IAMS research
   Code written by:
         Shannon Thornton                03 AUG 92

   Included in:  C_TREE.C, T_TREE.C, I_TREE.C, KS.PC

   Contains all data types used to model IAMS Knowledge Source.  The file
   consists of three parts:  column structures, table structures and index
   structures.  At the beginning of each part there is a BNF-like specification
   of the particular structure.

                                Naming Convention
   Functions used for manipulating the tree or list structure begin with the
   first letter of the structure used.  For example, to insert a column into
   the column tree, c_ins(..) is called.  The c_ indicates the function uses
   the column tree structure.  All functions that manipulate the table list
   inside a node in the column tree begin with c_t.  Any function that
   manipulates the index list inside the previously mentioned table list
   begins with c_i.  All table tree functions start with t_, and all index
   tree functions start with i_.  This convention is also used for global
   tree variables, c_tree, t_tree, and i_tree, which reference the column
   tree, table tree, and index tree, respectively.
*****************************************************************************/

#include "iams_const.h"            /* Include IAMS Constants file            */

/*****************************************************************************/
/* - - - - - - - - - - - - - - * * * * * * * * * * - - - - - - - - - - - - - */
/* - - - - - - - - - - - - - - * Column Structures * - - - - - - - - - - - - - */
/* - - - - - - - - - - - - - - * * * * * * * * * * - - - - - - - - - - - - - */
/*****************************************************************************
                  BNF-like Description of the Column Structures

  <Column>          ::=  <column name> <type> <size> [decimal]
                         { <table name> <key> <nullable> { <index name> } }
  <column name>     ::=  Any valid Oracle column name
  <table name>      ::=  Any valid Oracle table name
  <index name>      ::=  Any valid Oracle index name
  <type>            ::=   Char   | Date | Numeric |    Long
  <size>            ::=  1..240  |   7  |  1..38  | 0..65536
  [decimal]         ::=                | -37..37 |
  *** NOTE ***           Raw and Long Raw types are also acceptible.  The
                         Raw type is similar to Char, and the Long Raw is
                         similar to Long.  The same size restrictions apply.
                         See Oracle Database Administrator's Guide for more
                         information.
  <key>             ::=  <primary> <foreign> <composite>
  <primary>         ::=  'Y' | 'N' | ' '
  <foreign>         ::=  'Y' | 'N' | ' '
  <composite>       ::=  'Y' | 'N' | ' '
  <nullable>        ::=  'Y' | 'N'

  Conventions used in BNF-like syntax:
```

```
     { } - indicates the element(s) can occur zero or more times.
     < > - indicates the element is a non-terminal
     [ ] - indicates the element is a non-terminal that is optional in the
           definition
      |  - Synonym for the OR operator.  X | Y is read "X or Y"
     ' ' - Indicates character data is stored
*****************************************************************************/

struct c_idx_info {
   char name[NAMELENGTH];          /* Index name                              */
};

struct c_inode {                   /* Linked list node used in c_tab_info to */
   struct c_idx_info info;         /* store the indexes in which the current */
   struct c_inode *prev;           /* column is used in.                     */
   struct c_inode *next;
};

struct c_tab_info {
   char name[NAMELENGTH];          /* Table name                             */
   char key[4];                    /* PFC - Primary/Foreign/Composite        */
   char nullable;                  /* Y - Null, N - Not Null                 */
   struct c_inode *idx_head;       /* Pointer to a linked list of indexes in */
                                   /* which the column is used               */
   struct c_inode *idx_tail;       /* Pointer to the tail of the linked list */
                                   /* of tables.                             */
   struct c_inode *idx_cur;        /* Pointer to current node in the linked  */
                                   /* list.  Used for traversal.             */
};

struct c_tnode {                   /* Linked list node used in column to     */
                                   /* store the tables in which the current  */
                                   /* column is used.                        */
   struct c_tab_info info;         /* Data stored in the node                */
   struct c_tnode *prev;           /* Pointer to the previous node           */
   struct c_tnode *next;           /* Pointer to the next node               */
};

struct column {
   char name[NAMELENGTH];          /* Column name                            */
   char type[10];                  /* Column type                            */
   int size;                       /* Column size                            */
   int decimal;                    /* Number of decimal places if column is  */
                                   /* numeric                                */
   struct c_tnode *tbl_head;       /* Pointer to a linked list of tables in  */
                                   /* which the column appears               */
   struct c_tnode *tbl_tail;       /* Pointer to the tail of the linked list */
                                   /* of tables.                             */
   struct c_tnode *tbl_cur;        /* Pointer to current node in the linked  */
                                   /* list.  Used for traversal.             */
};

struct c_node {                    /* Tree node used to store information    */
                                   /* about each column                      */
   struct c_node *parent;          /* Pointer to the parent node             */
   struct column info;             /* Data stored in the node                */
```

```
   struct c_node *lchild;          /* Pointer to the left sub-tree          */
   struct c_node *rchild;          /* Pointer to the right sub-tree         */
};

/* ----------------------* End of Column Structures *---------------------- */


/*****************************************************************************/
/* - - - - - - - - - - - - - - * * * * * * * * * * * * - - - - - - - - - - - - - */
/* - - - - - - - - - - - - - * Table  Structures * - - - - - - - - - - - - - */
/* - - - - - - - - - - - - - * * * * * * * * * * * * - - - - - - - - - - - - - */
/*****************************************************************************
                  BNF-like Description of the Table Structures

  <Table>           ::=  <table name> { <column name> <key> <nullable> }
                         { <index name> <unique> }
  <column name>     ::=  Any valid Oracle column name
  <table name>      ::=  Any valid Oracle table name
  <index name>      ::=  Any valid Oracle index name
  <key>             ::=  <primary> <foreign> <composite>
  <primary>         ::=  'Y' | 'N' | ' '
  <foreign>         ::=  'Y' | 'N' | ' '
  <composite>       ::=  'Y' | 'N' | ' '
  <nullable>        ::=  'Y' | 'N'
  <unique>          ::=  'Y' | 'N'

  Conventions used in BNF-like syntax:
     { } - indicates the element(s) can occur zero or more times.
     < > - indicates the element is a non-terminal
     [ ] - indicates the element is a non-terminal that is optional in the
           definition
      |  - Synonym for the OR operator.  X | Y is read "X or Y"
     ' ' - Indicates character data is stored
*****************************************************************************/

struct t_idx_info {
   char name[NAMELENGTH];          /* Index name                            */
   char unique;                    /* Y - Unique, N - Nonunique             */
};

struct t_inode {                   /* Linked list node used in table to     */
                                   /* store the indexes used in the current */
                                   /* table.                                */
   struct t_idx_info info;         /* Data stored in the node               */
   struct t_inode *prev;           /* Pointer to the previous node          */
   struct t_inode *next;           /* Pointer to the next node              */
};

struct t_col_info {
   char name[NAMELENGTH];          /* Table name                            */
   char key[4];                    /* PFC - Primary/Foreign/Composite       */
   char nullable;                  /* Y - Null, N - Not Null                */
};

struct t_cnode {                   /* Linked list node used in table to     */
                                   /* store the columns used in the current */
```

Oct 11 17:46 1992 types.h Page 4

```
                                        /* table.                                  */
   struct t_col_info info;              /* Data stored in the node                 */
   struct t_cnode *prev;                /* Pointer to the previous node            */
   struct t_cnode *next;                /* Pointer to the next node                */
};

struct table {
   char name[NAMELENGTH];               /* Table name                              */
   struct t_cnode *col_head;            /* Pointer to a linked list of columns in  */
                                        /* the table                               */
   struct t_cnode *col_tail;            /* Pointer to the tail of the linked list  */
                                        /* of columns                              */
   struct t_cnode *col_cur;             /* Pointer to current node in the linked   */
                                        /* list.  Used for traversal.              */
   struct t_inode *idx_head;            /* Pointer to a linked list of indexes     */
                                        /* used by the table                       */
   struct t_inode *idx_tail;            /* Pointer to the tail of the linked list  */
                                        /* of indexes                              */
   struct t_inode *idx_cur;             /* Pointer to current node in the linked   */
                                        /* list.  Used for traversal.              */
};

struct t_node {                         /* Tree node used to store information     */
                                        /* about each table                        */
   struct t_node *parent;               /* Pointer to the parent node              */
   struct table info;                   /* Data stored in the node                 */
   struct t_node *lchild;               /* Pointer to the left sub-tree            */
   struct t_node *rchild;               /* Pointer to the right sub-tree           */
};

/* ----------------------* End of Table Structures *---------------------- */


/*****************************************************************************/
/* - - - - - - - - - - - - - - * * * * * * * * * * * * - - - - - - - - - - - - - */
/* - - - - - - - - - - - - - - * Index  Structures * - - - - - - - - - - - - - */
/* - - - - - - - - - - - - - - * * * * * * * * * * * * - - - - - - - - - - - - - */
/*****************************************************************************
                  BNF-like Description of the Index Structures

  <Index>           ::=  <index name> <unique> <table name> { <column name> }
  <column name>     ::=  Any valid Oracle column name
  <table name>      ::=  Any valid Oracle table name
  <index name>      ::=  Any valid Oracle index name
  <unique>          ::=  'Y' | 'N'

  Conventions used in BNF-like syntax:
     { } - indicates the element(s) can occur zero or more times.
     < > - indicates the element is a non-terminal
     [ ] - indicates the element is a non-terminal that is optional in the
           definition
      |  - Synonym for the OR operator.  X | Y is read "X or Y"
     ' ' - Indicates character data is stored
*****************************************************************************/

struct i_col_info {
```

```
   char name[NAMELENGTH];          /* Column name                          */
};

struct i_cnode {                   /* Linked list node used in index to    */
                                   /* store the columns used in the current*/
                                   /* index.                               */
   struct i_col_info info;         /* Date stored in the node              */
   struct i_cnode *prev;           /* Pointer to the previous node         */
   struct i_cnode *next;           /* Pointer to the next node             */
};

struct index {
   char name[NAMELENGTH];          /* Index name                           */
   char unique;                    /* Y - Yes, N - No                      */
   char tbl_name[NAMELENGTH];      /* Table name                           */
   struct i_cnode *col_head;       /* Pointer to a linked list of columns  */
                                   /* used in the index                    */
   struct i_cnode *col_tail;       /* Pointer to the tail of the linked list */
                                   /* of columns                           */
   struct i_cnode *col_cur;        /* Pointer to current node in the linked */
                                   /* list.  Used for traversal.           */
};

struct i_node {                    /* Tree node used to store information  */
                                   /* about each index                     */
   struct i_node *parent;          /* Pointer to the parent node           */
   struct index info;              /* Data stored in the node              */
   struct i_node *lchild;          /* Pointer to the left sub-tree         */
   struct i_node *rchild;          /* Pointer to the right sub-tree        */
};

/* ----------------------* End of Index Structures *---------------------- */
```

Waterways Experiment Station Cataloging-in-Publication Data

Wright, Peggy.
Intelligent access to multiple relational database systems / by Peggy Wright ; prepared for Discretionary Research Program, US Army Engineer Waterways Experiment Station.
175 p. : ill. ; 28 cm. — (Technical report ; ITL-92-9)
Includes bibliographical references.
1. Distributed data bases. 2. Computer interfaces. 3. Relational data bases. 4. Computers—Access control. I. US Army Engineer Waterways Experiment Station. II. Title. III. Series: Technical report (US Army Engineer Waterways Experiment Station) ; ITL-92-9.
TA7 W34 no.ITL-92-9